VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Detekce význačných částí obličeje

Face Landmarks Detection

2016 Kateřina Švédová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Kateřina Švédová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Detekce význačných částí obličeje
Face Landmarks Detection

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je detekce význačných částí obličeje v obrazu pomocí lokálních binárních příznaků. Výsledná aplikace bude vyznačovat nalezené časti obličeje do obrazu. Pro svou práci využijte aktuálního stavu poznání [3]. K trénování algoritmů strojoveho učení využijte datových množin, které jsou dostupné na internetu.

Ve své práci proveďte:
1. Nastudujte potřebnou literaturu pro detekci obličejů.
2. Implementujte vybraný algoritmus v jazyce C++ nebo Python.
3. Svou implementaci náležitě otestujte.
4. Proveďte závěrečné zhodnocení.

Seznam doporučené odborné literatury:

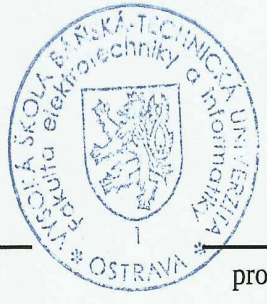
[1] Bradski, G.: Learning OpenCV: Computer Vision with the OpenCV Library, O'Reilly Media, ISBN: 978-0596516130
[2] Google C++ Code Style: https://google-styleguide.googlecode.com/svn/trunk/cppguide.html
[3] Ren, S., Cao, X., Wei Y., Sun, J.: Face Alignment at 3000 FPS via Regressing Local Binary Features, In Proceedings of Computer Vision and Pattern Recognition (CVPR), 2014, pp. 1685 - 1692, DOI: 10.1109/CVPR.2014.218

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Gaura, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 15. srpna 2016 ..... Šneblová

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 15. srpna 2016 ..... Šmídová .....

Zde bych ráda poděkovala Ing. Janu Gaurovi, Ph.D. za trpělivost, rady a návrh tématu diplomové práce, Ing. Petru Strakošovi, Ph.D. za konzultace týkající se MatLabu, Bc. Martinu Besedovi za trpělivost a podporu při vzniku této práce, Ing. Davidu Horákovi, Ph.D. za kontrolu matematické části, Bc. Renatě Plouharové za konzultaci týkající se překladu odborných článků a v neposlední řadě také rodině a přátelům.

Abstrakt

Tato diplomová práce se zabývá implementací algoritmu Local Binary Features v Pythonu. Obsahem práce je představení detekování význačných bodů v obličeji, následné otestování a porovnání s implementací v MatLabu.
Práce je rozdělena do několika bloků. Ve druhé až šesté kapitole podrobněji rozebírám funkčnost algoritmů a možnosti jejich implementace. V sedmé kapitole se zabývám samotnou implementací a v osmé kapitole naleznete experimenty, tedy praktické odzkoušení a zhodnocení aplikace.
Cílem práce bylo srovnání implementace algoritmu Local Binary Features v MatLabu a Pythonu. Získané výsledky ukazují, že implementace v Pythonu, přiložená k této práci, není vhodná pro reálné využití vzhledem k časové náročnosti, ačkoli po stránce schopnosti detekovat význačné rysy v obličeji je stejně kvalitní jako implementace v MatLabu.

Klíčová slova: Detekce význačných rysů, Lokální rysy obličeje, Analýza obrazu, Python

Abstract

This thesis deals with the implementation of the algorithm Local Binary Features at Python. Main topic of this thesis is face landmarks detection, testing and alignment with implementation in MatLab.
The work is divided into several blocks. From the second to the sixth chapter I am describing the algorithms and their possible implementations in detail. The seventh chapter deals with the implementation itself and the eighth chapter contains experiments, thus a practical test and evaluation of the application.
The main focus of this thesis was a comparison of two implementations of Local Binary Features algorithm - in MatLab and Python. Results show, that the implementation in Python, attached to this thesis, is not suitable for real-life application, because it is very time-demanding. However, considering its ability to detect face landmarks, it is the same quality MatLab implementation.

Key Words: Face landmarks detection, Local Binary Features, Image analysis, Python

Obsah

Seznam použitých zkratek a symbolů

LBF	–	Local Binary Features
RF	–	Random Forest
CPR	–	Cascade Pose Regression
PCA	–	Principal Component Analysis
AAM	–	Active Appearance Models
ASM	–	Active Shape Models

Seznam obrázků

Seznam tabulek

Seznam pseudokódů

Seznam výpisů zdrojového kódu

1 Úvod

Tato diplomová práce se zabývá implementací metody Local Binary Features (dále LBF) pro detekci význačných částí obličeje (obočí, oči, nos, rty, hrana obličeje) v Pythonu. Během implementace bylo použito několik algoritmů pro zpracování dat, mezi něž patří např. Random Forest (dále RF), globální regrese a globální predikce.
Hlavním cílem této práce je implementace a otestování algoritmu Local Binary Features a jeho srovnání s volně dostupnou implementací v MatLabu [5].

V kapitole 2 naleznete krátký úvod do problematiky detekce obličeje. Kapitola 3 se zabývá problematikou Cascaded Pose Regression, následující kapitola 4 rozebírá princip porovnání tváří pomocí algoritmu Explicit shape regression a kapitola 5 popisuje algoritmus Local Binary Features. Kapitola 6 krátce popíše využité pomocné funkce a následující kapitola 7 rozebírá postup implementace výsledného algoritmu. Závěrem kapitola 8 obsahuje zhodnocení výsledků a porovnání s volně dostupnou implementací v Pythonu.

1.1 Analýza obrazu

Analýza obrazu je jedním z aktuálních témat. Díky zvyšujícímu se výkonu výpočetní techniky, ale také její dostupnosti, se začíná více projevovat nejen v životě běžných uživatelů. Mnohé algoritmy začínají být díky těmto aspektům dostupnější. V reálném životě se můžeme setkat s využitím na různých místech, např. U bezpečnostních kamer na veřejných místech nebo na různých sociálních sítích, kde se často využívá detekce obličeje (Facebook, Google+ apod.).

1.2 Local Binary Features

Tento algoritmus využívá dvou hlavních částí - sadu binárních lokálních příznaků a lokální princip pro naučení detekce příslušných rysů.
V první fázi, tzv. trénování, nám lokální princip umožňuje učit binární znaky pro každý význačný bod nezávisle na ostatních. Získané lokální binární znaky poté spojíme a využijeme k naučení pomocí lineární regrese. Těmito kroky je splněno natrénování algoritmu a jsou získány potřebné proměnné pro následné testování.
Ve druhé fázi, tedy testování, využijeme globální predikce neboli odhadu k určení znaků s využitím již zmíněných dat, které jsme získali z první fáze.

2 Detekce obličeje

Jednou z mnoha problematik, kterou se zabývá počítačové vidění, je detekce obličeje. Důsledkem dostupnosti a zvyšujícímu se výkonu výpočetní techniky může být detekce obličeje využita již téměř ve všech přístrojích, které mají snímací zařízení (osobní počítač, mobilní telefon, tablet aj.). Bezpečnostní složky mohou tuto problematiku výrazně využívat, zejména při hledání pachatele ze záznamu bezpečnostní kamery, kde na základě porovnání jeho obličeje s databází jsou schopni zájmovou osobu identifikovat.

Detekce obličeje se provádí na základě analýzy vstupního obrazu. Metody nalezení obličeje se dají rozdělit podle přístupu následovně [9]:

1. Vědomostně založené modely - využívají se především pro lokalizaci obličeje. Jsou založené na sadě pravidel označujících typický obličej. Tyto pravidla zachycují vztahy mezi typickými znaky obličeje (vzdálenost očí apod.)

2. Hledání neměnných znaků obličeje - tyto metody jsou opět spíše vhodné pro lokalizaci obličeje. Hledají typické znaky, které se nemění při otočení hlavy, změně osvětlení apod.

3. Porovnání se vzory - tento přístup je vhodný pro lokalizaci i detekci. Na základě předem definovaných funkcí (manuálně nebo parametricky), které popisují obličej nebo jeho části, spočteme korelaci se vstupním obrazem. Na základě výsledku korelace je detekován obličej.

4. Vzhledově založené metody - metoda vhodná zejména pro detekci obličeje. z databáze obličejů, obsahující nejrůznější vzhledy obličeje, se vytvoří model na jehož základě se provádí samostatná detekce.

Úkolem detekce obličeje v obraze je zjistit, zda-li se objekt v obraze nachází a poté určit oblast, kde se hledaný objekt nachází. Nalezení obličeje, ale může být ztíženo několika faktory:

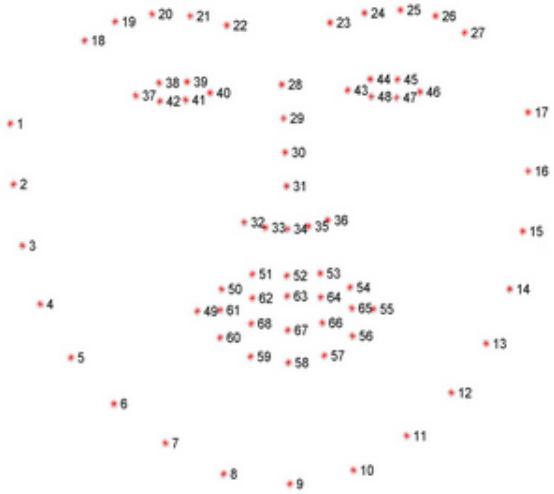
- Pozice kamery vzhledem k obličeji
- Absence nebo přítomnost význačných prvků obličeje
- Výraz v obličeji
- Zakrytí části obličeje
- Osvětlení

Proto je nutné si při volbě algoritmu detekce obličeje rozmyslet, zda bude vstupní obraz těmito faktory ovlivněn či nikoliv (např. půjde o průkazové foto - tedy pohled bude zepředu a bude mít dobře osvětlenou tvář nebo to bude tzv. momentka, u které je velká pravděpodobnost, že v obličeji bude stín, bude libovolně natočen nebo částečně zakryt).

2.1 Použitá detekce obličeje

Existuje mnoho algoritmů pro detekování obličeje. Jedním z nich je například velmi známý algoritmus Viola-Jones [26]. Ten využívá kaskády slabých klasifikátorů s aplikací jednoduchých Haarových znaků. Je to nejčastěji používaný algoritmus, jehož základní implementace je součástí OpenCV.

V této práci využijeme toho, že algoritmus LBF trénujeme i testujeme na datasetech[1], které mají již ručně označené význačné znaky. Díky tomu nemusíme použít algoritmus pro detekci obličeje. Vezmeme příslušný `*.pts` soubor, který obsahuje souřadnice 68-mi bodů, jež označují význačné znaky v obličeji. Nalezením krajních hodnot pro x-ovou i y-ovou souřadnici označíme oblast, kde se obličej nachází. Dále pracujeme pouze v dané oblasti, kde kromě ručně označených znaků máme k dispozici ještě masku znázorňující rozmístění význačných bodů v obličeji. Maska je zobrazena na Obr. 1.



Obr. 1: Rozmístění význačných bodů v obličeji [1]

[1] `http://ibug.doc.ic.ac.uk/resources/facial-point-annotations/`

3 Cascaded Pose Regression

V [3] autoři představují rychlý a přesný algoritmus pro výpočet 2D pozice objektu v obraze nazvaný Cascaded Pose Regression (dále jen CPR). CPR postupně zpřesňuje volné počáteční odhady, kde se každé zpřesnění provádí pomocí jiného regresoru. Každý regresor provádí jednoduché měření obrazu, které závisí na výstupu předchozího regresoru. Celý systém je automaticky naučen z trénovacích dat určených člověkem. CPR není omezeno pevnými transformacemi: vzhled objektu je parametricky rozdílný pro znaky objektů tak, aby postupně bez omezení tvarovalo a spojovalo objekty.

Detekce a lokalizace jsou funkce potřebné v počítačovém vidění. Pomocí detekce jsme schopni zodpovědět otázku: „Je objekt x v obrázku?“. Lokalizace je poté více komplexnější problém - ve své nejrozšířenější a nejjednodušší podobě lokalizace představuje identifikaci nejmenších obdélníkových oblastí v obraze, které obsahují dané objekty. Toto je naprosto dostačující pro kategorie, kdy chceme získat pohled na tvář zepředu a je nutné obraz přepočítat a upravit.

Hlavní postup pro lokalizaci objektu a jeho rozsahu je použití „sliding window“ nebo-li „klouzavého okna“, to znamená opakování binární klasifikační úlohy „Je objekt x v lokaci y?“ pro co nejpřesnější určení, tedy pro jemné vzorkování obrazu. Ačkoli tímto získáme vysoký počet testovacích oken, můžeme metodu sliding window zefektivnit pomocí kaskády [17], vzdáleností transformace [18], vyhledávání „brunch and bound“ [19] a zjemnění [20].

Tyto metody mohou být rozšířeny komplexněji pomocí opakovaného odpovídání na dotaz „Je objekt x v lokaci y s pozicí θ?“, kde pro každou pozici θ máme jednu otázku. Například pro detekci obličeje se obvykle trénuje samostatný klasifikátor pro různé úrovně natočení [17]. Tato cesta vede ke kombinatorické explozi otázek a ačkoli vhodná vyhledávací strategie může pomoct, nakonec takový přístup nemůže škálovat komplexnější pozice.

Je-li dán hrubý odhad lokalizace objektu, pak algoritmus přímo odpovídá na otázku „Jaká je pozice θ objektu x?“ tak, že rekonstruuje pozici bez vykonání potenciálně výpočetně náročného a rozvětveného hledání. V podstatě přesně toto realizují standardní regresní techniky. Ačkoli pro určité úkoly počítačového vidění regrese bývá úspěšná, její použitelnost k více obecnému odhadu pozice zůstává nejasná.

Dále autoři navrhují při zesílené regresi k učení pevně lineární nebo kaskádovou sekvenci slabých regresorů - v jejich případě náhodný fern. Klíčový rozdíl oproti předchozímu přístupu je použití *pose-indexed* znaků [21]: znaky jejichž detekce závisí na datech obrazu a současně na odhadu pozice. Za předpokladu slabé invariance pose-indexed znak odvodíme klíčovým algoritmem pro pozici odhadu nazvaného CPR. Přesný model může být naučen s překvapivě malým množstvím

dat (zhruba 100 označených trénovacích dat). CPR je rychlé, přesné a jednoduše trénovatelné na různých objektech.

Aby byla možnost jasné diskuze o pozici a vzhledu objektů, tak autoři [3] předpokládají, že existuje neznámý model vytvoření obrazu $G : \mathcal{O} \times \Theta \to I$, který má vzhled objektu $o \in \mathcal{O}$, pozici $\theta \in \Theta$ a vytváří obraz $I \in \mathcal{I}$. Nikdy nemáme přímý přístup k G nebo o, nicméně jsou nezbytné pro následující odvození. Například můžeme napsat $I_{\theta_1} = G(o, \theta_1)$ a $I_{\theta_2} = G(o, \theta_2)$ k označení dvou obrazů stejného objektu o ve dvou pozicích θ_1 a θ_2. Předpokládejme, že $G(o_1, \theta_1) = G(o_2, \theta_2)$, pokud $o_1 = o_2$ a $\theta_1 = \theta_2$, jinak není možné, aby odhad pozice byl jedinečný. Požadujeme, aby Θ společně s operací $\circ$ vytvořilo grupu. Jestliže máme dvě pozice θ_1, θ_2, pak zapíšeme jako $\theta = \theta_1 \circ \theta_2$ k označení nové pozice vytvořené kombinací θ_1 a θ_2, $\bar{\theta}$ k označení inverzního prvku θ a e k popisu neutrálního prvku. K odhadu relativní chyby mezi dvěmi pozicemi, použijeme funkce $d : \Theta \times \Theta \to \mathbb{R}$, kde $d(\theta_1, \theta_2)$ může záviset pouze na relativní pozici $\bar{\theta}_1 \circ \theta_2$ nebo ekvivalentně taky $d(\theta_\delta \circ \theta_1, \theta_\delta \circ \theta_2) = d(\theta_1, \theta_2)$ pro všechny $\theta_1, \theta_2, \theta_\delta \in \Theta$.

3.1 Pose-Indexed znaky a slabá invariance

Již bylo zmíněno, že během této práce spoléháme na pose-indexed znaky. Pose-indexed znak je jednoduchá funkce vytvořena z $h : \Theta \times \mathcal{I} \to \mathbb{R}$. Řekněme, že h je slabě invariantní, když $\forall \theta, \theta_\delta \in \Theta$ platí:

$$h(\theta, G(o, e)) = h(\theta_\delta \circ \theta, G(o, \theta_\delta)), \tag{1}$$

nebo ekvivalentně, h je slabě invariantní, pokud $\forall \theta_1, \theta_2, \theta_\delta \in \Theta$ platí:

$$h(\theta_1, G(o, \theta_2)) = h(\theta_\delta \circ \theta_1, G(o, \theta_\delta \circ \theta_2)). \tag{2}$$

Je snadné dokázat, že rovnice (2) platí právě tehdy, když platí rovnice (1). Dalším způsobem vyjádření předchozích rovnic je, že $h(\theta_1, G(o, \theta_2))$ závisí pouze na objektu o a relativní pozici $\bar{\theta}_1 \circ \theta_2$ mezi vstupní pozicí θ_1 a skutečnou pozicí θ_2. Dále je h slabě invariantní, když její funkční hodnotou je konstantně daný ucelený odhad pozice.

Lze si všimnout, že invariance definována výše je mnohem slabší požadavek než obecně pozice invariance, které by mohly být uvedeny jako: $h(G(o, \theta)) = h(G(o, \theta_\delta \circ \theta))$. Návrh invariantní funkce h, který postačuje pozdější definici je mimořádně složitý, ale použité definici postačí požadavek invariance pouze, když je dán pevně odhad pozice.
Jak je uvedeno v [3], slabý předpoklad invariance potvrdí derivace, které umožňují dokázat silnou konvergenci pro výsledné algoritmy.

Pose-indexed Control Point Features

Ve všech experimentech uvedených v [3], používají extrémně jednoduchý a rychle vypočítatelný Control Point Features. V implementaci je každý kontrolní bod spočten jako rozdíl dvou pixelů obrazu v předdefinované lokaci. Respektive, každý znak $h_{p1,p2}$ je definován pomocí dvou lokací p_1 a p_2 a je vyhodnocen výpočtem $h_{p_1,p_2}(I) = I(p_1) - I(p_2)$, kde I_p označuje hodnotu ze škály šedi v obraze I v lokaci p.
Kromě rychlosti a překvapivé účinnosti v reálných aplikacích, je výhodou uvedených znaků jejich přímá indexace pozice. Například předpokládejme, že pozice objektu je určena pomocí translace, rotace, rozsahu a orientace (nebo dalších parametrů). Pro každou pozici θ můžeme definovat přidruženou $3\times$ 3 homografní matici H_θ, přesněji lokaci p v homogenních souřadnicích a definovat $h_{p_1,p_2}(\theta, I) = I(H_\theta p_1) - I(H_\theta p_2)$. Následně není obtížné rozšířit tento postup ke složeným objektům, pokud každá část má přiřazenou vlastní homografní matici.

3.2 Cascaded Pose Regression

Tato podkapitola popisuje vyhodnocení a trénovací procedury pro kaskádový regresor pozic $R = (R^1, \ldots, R^T)$ (algoritmus 1 resp. algoritmus 2, [3]).

Algoritmus 1 Vyhodnocení Cascaded Pose Regression

Input: Obraz I, počáteční pozice θ^0
1: **for** t = 1 to T **do**
2: $x = h^t(\theta^{t-1}, I)$ // vypocitani priznaku
3: $\theta_\delta = R^t(x)$ // vyhodnoceni regresoru
4: $\theta^t = \theta^{t-1} \circ \theta_\delta$ // aktualizace pozice θ^t
5: **end for**
Output: θ^T

Budeme trénovat kaskádový regresor $R = (R^1, \ldots, R^T)$ tak, že je dána vstupní pozice θ^0, $R(\theta^0, I)$ je poté vyhodnocen výpočtem:

$$\theta^t = \theta^{t-1} \circ R^t(h^t(\theta^{t-1}, I)), \tag{3}$$

$t = 1, \ldots, T$ a finálním výstupem je θ^T. Každý regresor R^t je trénován se snahou snížit rozdíl mezi skutečnou pozicí a spočítanou pozicí pomocí předchozích komponent použitím (pose-indexed) znaku h^t. Cílem je optimalizovat následující ztráty:

$$\mathcal{L} = \sum_{i=1}^{N} d(\theta_i^T, \theta_i). \tag{4}$$

Začneme výpočtem $\theta^0 = \arg\min_\theta \sum_i d(\theta, \theta_i)$ a označíme $\theta_i^0 = \theta^0$ pro každé i. θ^0 je jedinečná pozice odhadu stanovená danou nejnižší trénovací chybou aniž by závisela na dalších regresorech.

Dále popíšeme proces trénování R^t daného jako $(R^1, \ldots, R^{t-1})$. V každé fázi t, začneme trénovat náhodně vygenerovaným pose-indexed znakem h^t a výpočtem $x_i = h^t(\theta_i^{t-1}, I_i)$ pro každý trénovací příklad I_i s předchozím odhadem pozice θ_i^{t-1}. Cílem je naučit regresor R^t tak, že $\theta_i^t = \theta_i^{t-1} \circ R^t(x_i)$ minimalizuje ztráty v rovnici (4). Po několika iteracích můžeme psát:

$$R^t = \arg\min_R \sum_i d(R(x_i), \tilde{\theta}_i), \tag{5}$$

kde $\tilde{\theta}_i = \bar{\theta}_i^{t-1} \circ \theta_i$. Můžeme vyřešit R^t s použitím standardních regresních technik. Navíc před R^t potřebujeme pouze lehce snížit chybu, proto můžeme trénovat odděleně jednotlivé regresory pro každou souřadnici $\tilde{\theta}$ a poté jednoduše uchovat ten nejlepší. V této práci spoléháme na náhodnou regresi fern (kapitola 3.2.1).
Po trénování R^t aplikujeme rovnici (3) k výpočtu θ_i^t pro využití v další fázi trénování. Když je regresor R^t neschopen dále redukovat chybu trénování, tak skončíme. Nechť:

$$\epsilon_t = \sum_i d(\theta_i^t, \theta_i) / \sum_i d(\theta_i^{t-1}, \theta_i). \tag{6}$$

Pokud je $\epsilon_t \geq 1$, trénování ukončíme jinak pokračujeme v trénování po T krocích nebo dokud chyba nepoklesne pod určitou hodnotu. Celý tréninkový proces je znázorněn jako algoritmus 2.

Algoritmus 2 Trénování Cascaded Pose Regression

Input: Data (I_i, θ_i), pro $i = 1 \ldots N$
1: $\theta^0 = \arg\min_\theta \sum_i d(\theta, \theta_i)$
2: $\theta_i^0 = \theta^0$ pro $i = 1 \ldots N$
3: **for** t = 1 to T **do**
4: $x_i = h^t(\theta^{t-1}, I_i)$
5: $\tilde{\theta}_i = \bar{\theta}_i^{t-1} \circ \theta_i$
6: $R^t = \arg\min_R \sum_i d(R(x_i), \tilde{\theta}_i)$
7: $\theta_i^t = \theta_i^{t-1} \circ R^t(x_i)$
8: $\epsilon_t = \sum_i d(\theta_i^t, \theta_i) / \sum_i d(\theta_i^{t-1}, \theta_i)$
9: If $\epsilon_t \geq 1$ stop
10: **end for**
Output: $R = (R^1, \ldots, R^T)$

3.2.1 Náhodný Fern Regresor

V každé fázi kaskádově trénujeme náhodný fern regresor. Fern regresor vezme vstupní vektor $x_i \in \mathbb{R}^F$ a vytvoří výstup $y_i \in \mathbb{R}$. Toto je vytvořeno náhodným vybíráním prvků S z F-dimenzionálního vektoru znaků s nahrazením a poté souboru náhodných hranic S. Každé x_i skončí v jedné z 2^F *bin*. Odhad y pro *bin* je průměrem z y_i trénovacích příkladů, které skončily v dané *bin*. V každé fázi kaskády vybereme nejlepší fern z pohledu trénovací chyby z mnohých náhodně generovaných fern R.

3.2.2 Shlukování pozic

Záleží na nastavení před aplikováním CPR, ale stává se, že algoritmus výjimečně selže při odhadu korektní pozice. Nicméně častější je opětovné spuštění CPR s rozdílnou počáteční pozicí, což přináší přijatelnější odhad. Proto bylo v [3] použito jednoduchého shlukování pozic, které dokáže vylepšit algoritmus. Pro každý obraz bylo CPR spuštěno K-krát s různou počáteční pozicí. Po proběhnutí všech K pokusů, je vybrána pozice v oblasti s největší hustotou všech pozic, jakožto finální předpověď algoritmu, pomocí použití „Parsen“ okna s Gaussiánovým jádrem šířky 1 (použití normalizované vzdálenosti) k odhadu hustoty každé pozice pro srovnání s dalšími $K-1$ pozicemi.

3.2.3 Rychlost konvergence

V [3] autoři ověřují, že jejich iterační schéma bude konvergovat pod dostatečně slabými předpoklady a mimo jiné ukáží, že rychlost konvergence je exponenciální při slabých chybách z komponentních regresorů. Důkaz je v podstatě velmi podobný důkazům pro konvergenci zesíleného regresoru, ale požaduje představu o problematice slabé naučitelnosti.

Nechť h představuje sadu standardních (ne pose-indexed) znaků. Definujme relativní chybu regresoru R na datasetu (I_i, θ_i) jako $\epsilon = \sum_i d(R(h(I_i)), \theta_i) / \sum_i d(\theta, \theta_i)$. Dokáže-li R více než jen poskytnout jednotnou odpověď θ, pak $\epsilon < 1$. Zcela zřejmá konvergence dokazuje, že CPR i zesílený regresor požadují, abychom měli přístup ke slabému učení, které daný dataset (I_i, θ_i) dovede k výstupu regresoru s relativní chybou $\epsilon \leq \beta$ pro nějaké $\beta < 1$. Za těchto podmínek, rychlost konvergence pro oboje, CPR i zesílený regresor, je dána pomocí $\epsilon^T \leq \beta^T$.

Základní rozdíl mezi konvergencí CPR a zesíleného regresoru spočívá v síle slabé naučitelnosti předpokladů. Nechť $I_i = G(o_i, \theta_i')$ pro nějaké neznámé o_i. V CPR potřebujeme přístup ke slabě naučitelnému objektu, takže může být výstup regresoru s $\epsilon \leq \beta$ na trénovacím setu (I_i, θ_i) pouze když $\theta_i = \theta_i'$. Pro zesílenou regresi potřebujeme také přístup ke slabě naučitelnému objektu, takže může být výstup regresoru s $\epsilon \leq \beta$ na libovolný trénovací set (I_i, θ_i), kde θ_i potřebuje rozdílné θ_i'. V praxi je základní hodnota β mnohem menší u CPR, ačkoli oba, CPR i zesílená regrese, konvergují exponenciálně je-li splněna podmínka slabé naučitelnosti.

3.3 Navýšení dat

Použití pose-indexed znaků může uměle simulovat vysoké množství dat z N trénovacích příkladů pomocí jednoduchého použití rozdílných počátečních odhadů pozice. Když potřebujeme sledovat každý objekt v každé pozici použití těchto znaků umožní zabránit kombinatorické explozi dat. Předpokládáme, že jsou dané trénovací sety (I_i, θ_i) pro $i = 1, \ldots, N$ a požadujeme k simulaci dodatečná data. Připomeňme, že každý obraz I_i je generován použitím $G(o_i, \theta_i)$, kde o_i je neznámý znak objektu. Použitím trénovacích dat můžeme odhadnout distribuci $\mathcal{D}$ pro pozici θ_i (nebo použitím jejich empirických distribucí). V [3] by autoři chtěli použít $\mathcal{D}$ k rozšíření trénovacího setu pomocí vzorkování $\theta_j \sim \mathcal{D}$ a vygenerování nových trénovacích obrazů $I_{ij} = G(o_i, \theta_j)$. Ačkoli nemáme přístup k G nebo o_i, tak můžeme aktuálně docílit identického efektu pomocí využití pose-indexed příznaků, které jsou slabě invariatní. Důkaz a formalizaci tohoto tvrzení najdete v [3], na straně 5.

4 Porovnání tváří pomocí Explicit Shape Regression

Porovnání tváří nebo lokalizování význačných částí obličeje jako jsou oči, nos, rty a hrana obličeje, je nezbytné pro úkoly při rozpoznání obličeje, sledování tváří, obličejových animací a 3D modely obličejů. s prudkým nárůstem osobních a webových fotografií je žádaný plně automatický, efektivní a masivní algoritmus pro porovnání tváří. Tyto požadavky jsou stále výzvou pro nejnovější metody, protože fotografie se liší otočením tváře, osvětlením nebo zakrytím části obličeje.

Tvar obličeje $s = [x_1, y_1, \ldots, x_{N_{fp}}, y_{N_{fp}}]$ je složený z N_{fp} význačných bodů v obličeji. Daný obraz obličeje a cíl pro porovnání obličeje dává odhad tvaru S tak, že je přiblížíme co nejblíže skutečnému tvaru $\hat{\mathrm{S}}$ minimalizací:

$$\| s - \hat{s} \|_2 \tag{7}$$

Chyba porovnání v rovnici (7) je obvykle použita jako směrnice pro trénování a zhodnocení výkonu. Ale během testování to nejsme schopni minimalizovat, protože $\hat{S}$ je neznámá. Podle toho jaký je odhad S můžeme většinu srovnávacích metod klasifikovat do dvou kategorií: *optimization-based* a *regression-based*.

Optimization-based metody minimalizují další chyby vzniklé z rovnice (7). *Regression-based* metody učí regresivní funkci tak, že mapují vzhled obrazu k cílového výstupu. Komplexní varianty jsou naučeny z rozsáhlých trénovacích dat a testování je obvykle efektivní. Více o těchto metodách se můžete dočíst v [2].

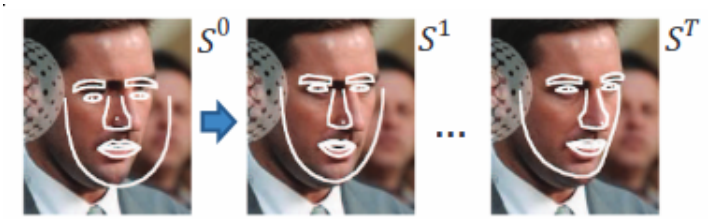
Tvarové omezení je nezbytné ve všech metodách. Jenom málo význačných bodů v obličeji (např. zorničky, koutky úst) můžeme spolehlivě charakterizovat pomocí vzhledu obrazu. Mnoho dalších nevýznačných bodů (body podél obrysu tváře) potřebují upřesnit z tvarového omezení - korelací mezi význačnými body. Mnoho existujících prací používá parametrický model tvaru k vynucení těchto omezení, např. PCA model v AAM [14] a ASM [15].

Navzdory úspěšnosti parametrického tvaru modelu, tvarování modelu je často předurčené. Použití pevného tvaru modelu v opakujícím se porovnávacím procesu může být jenom částečně optimální. Například v počátečním stavu (tvar je daleko od skutečného cíle) je výhodné použít omezený model pro rychlou konvergenci a lepší regularizaci, v pozdějším stavu (tvar zhruba odpovídá skutečnosti) můžeme chtít použít více flexibilní model tvaru s větší ohebností pro zpřesnění. Nicméně znalosti tohoto přizpůsobení tvaru modelu jsou využity velmi vzácně.

Metoda *Explicit Shape Regression* ukazuje způsob na základě regression-based přístupu bez použití parametrických modelů tvarů. Regresor je trénován jednoznačnou minimalizací srovnávací chyby přes trénovací data a všechny význačné body jsou regresně spojeny ve vektorovém výstupu.

Použitý regresor v [2] realizuje tvarové omezení v neparametrickém způsobu: regresní tvar je vždy lineární kombinací ze všech trénovacích tvarů. Je více rozlišující použít příznaky přes obraz pro všechny význačné body než použít lokální části pro individuální význačné body. Tyto vlastnosti umožňují učit flexibilně model z rozsáhlých trénovacích dat.

Spojení regrese veškerých tvarů pro rozsáhlé obrazové variace je v současnosti výzvou. V [2] autoři navrhují zesílený regresor k postupnému odvození tvaru - prvotní regresory se zabývají rozsáhlými změnami tvarů a zajištění odolnosti, zatímco pozdější regresory se zabývají malými tvarovými změnami a zajišťují přesnost. Omezení tvaru je automaticky adaptivně vynuceno z nejhrubšího odhadu k lepšímu určení tvaru. Ilustraci najdete na obrázku 2, detailní popis poté v sekci 4.1.2



Obr. 2: Explicit shape regression [2]

4.1 Porovnání tváří pomocí regrese

Používáme zesílený regresor [13] ke spojení T slabých regresorů $(R^1, \ldots, R^t, \ldots, R^T)$ pomocí součtu. Mějme daný obraz obličeje I a počáteční tvar obličeje S^0 (počátečním tvarem může být jednoduše průměr tvaru, více informací najdete v [2], kapitole 3), každý regresor vypočítá zvýšení tvaru o δS z obrazových znaků a poté aktualizuje tvar obličeje. Tato aktualizace se provádí způsobem kaskády:

$$S^t = S^{t-1} + R^t(I, S^{t-1}), \quad t = 1, \ldots, T, \tag{8}$$

kde t-tý slabý regresor R^t aktualizuje předchozí tvar S^{t-1} k novému tvaru S^t.

Všimněme si, že regresor R^t závisí jak na obrazu I, tak i na předchozím odhadu tvaru S^{t-1}. Toto popíšeme v kapitole 4.2, použijeme tzv. *Shape-indexed znaky (obrazu)*, kde se vzájemně učí každý R^t pomocí předchozího odhadu. Takové znaky můžou značně zlepšit zesílený regresor dosažením geometrické neměnnosti. Podobná myšlenka je použita v [3] nebo v kapitole 3.

[2] `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/cvpr12_facealignment.pdf`

Při N trénovacích příkladech $\{(I_i, \hat{S}_i^{t-1})\}_{i=1}^N$ se regresory $(R^1, \ldots, R^t \ldots, R^T)$ učí po částech, dokud není dostatečně snížena trénovací chyba. Každý regresor R^t je učen pomocí jasně dané minimalizace součtu srovnávacích chyb rovnice (7) do té doby než

$$R^t = \arg\min_{\mathrm{R}} \sum_{i=1}^{N} \parallel \hat{S}_i - (S_i^{t-1} + R(I_i, S_i^{t-1})) \parallel \tag{9}$$

kde S_i^{t-1} je odhad z předchozí fáze.

4.1.1 Dvouúrovňová kaskádová regrese

Dřívější metody používají jednoduché slabé regresory jako nalezení kořene [22] nebo fern [3] v podobně zesílených regresorech. Nicméně při experimentech předcházející [2] našli autoři takové regresory, které jsou také slabé, výsledky trénování pomalu konvergují a mají nízký výkon při testování. Jednoduchý slabý regresor může pouze velmi málo snížit chybu.

Toto zjištění je zásadní pro učení dobrého slabého regresoru, který může rychle redukovat chybu. Proto v [2] autoři k učení navrhují, aby se slabý regresor R^t učil pomocí dvouúrovňové zesílené regrese, tedy $R^t = (r^1, \ldots, r^k, \ldots, r^K)$. Tento problém je podobný jako v rovnicích (8), (9), ale významným rozdílem je, že shape-indexed znaky obrazu jsou pevně dány v druhé úrovni, tedy jsou indexovány pouze relativně od S^{t-1}. Je důležité, že ačkoli každý regresor r je spíše slabý a dovoluje znakům měnit indexaci, tak je často nestálý. Také pevně daný znak může vést k mnohem rychlejšímu trénování.

4.1.2 Primitivní regresor

Použijeme *fern* jako primitivní regresor r. Fern byl nejdříve představen pro klasifikaci [11] a později použit pro regresi [3]. Fern je sestaven z F znaků a hranic, které dělí prostor znaků (a všech trénovacích příkladů) do 2^F *bin*. Každá *bin* b je přiřazená k regresnímu výstupu δS_b, který minimalizuje srovnávací chybu trénovacích chyb Ω_b *bin*:

$$\delta S_b = \arg\min_{\delta \mathrm{S}} \sum_{i \in \Omega_b} \parallel \hat{S}_i - (S_i + \delta S) \parallel, \tag{10}$$

kde S_i označuje odhad tvaru v předchozí fázi. Řešením pro rovnici (10) je střední hodnota rozdílů tvarů:

$$\delta S_b = \frac{\sum_{i \in \Omega_b} (\hat{S}_i - S_i)}{|\Omega_b|}. \tag{11}$$

K překonání přetečení v případě nedostačujících trénovacích dat v *bin*, je nastaveno snížení chyby následovně:

$$\delta S_b = \frac{1}{1+\beta/|\Omega_b|}\frac{\sum_{i\in\Omega_b}(\hat{S}_i - S_i)}{|\Omega_b|}, \tag{12}$$

kde β je libovolný snižovací parametr. Pokud má *bin* dostatek trénovacích příkladů, β vytvoří malý efekt, jinak adaptivně redukuje odhad.

Neparametrické omezení tvaru

Pomocí učení vektorového regresoru a explicitně minimalizovaného tvaru srovnávací chyby rovnice (7) je uchována korelace mezi souřadnicemi tvaru. Protože každý tvar je upraven součtem jako v rovnici (8), a každý tvar je zvýšen lineární kombinací určitých trénovacích tvarů $\{\hat{S}_i\}$ jako v rovnici (11) nebo v rovnici (12), tak lze snadno zjistit, že finální regresní tvar S může být vyjádřen jako počáteční tvar S^0 plus lineární kombinace přes všechny trénovací příklady:

$$s \ = S^0 + \sum_{i=1}^{N} \omega_i \hat{S}_i. \tag{13}$$

Dokud vyhovuje počáteční tvar S^0 tvarovému omezení, tak regresní tvar vždy spočívá v omezení v lineárním subprostoru konstruovaném přes všechny trénovací tvary. Střední tvar v regresi tedy také vyhovuje omezení. Při srovnání s tvarem modelu PCA, dojdeme ke zjištění, že neparametrické tvarové omezení je deterministicky přizpůsobivé během učení.

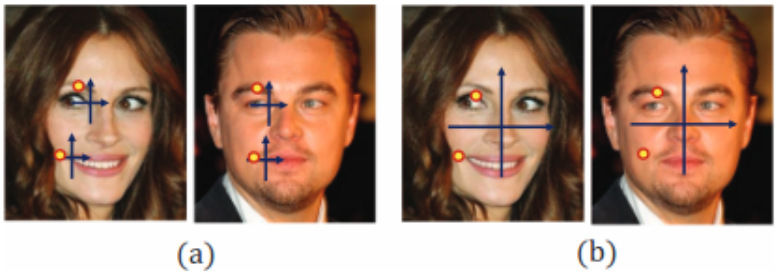
4.2 Shape-indexed znaky obrazu

Pro efektivní regresi použijeme prostý rozdíl znaků, tzn. intenzitu rozdílu dvou pixelů v obrazu. Některé znaky jsou extrémně nenáročné k výpočtu a docela efektivně dávají dostačující trénovací data. Pixel je indexován relativně k aktuálnímu odhadu tvaru přesněji než k souřadnicím originálního obrazu. Podobná myšlenka je také v [3]. Tento přístup dosahuje nejlepší geometrické neměnnosti a vede k nejjednodušším regresním problémům a nejrychlejší konvergenci při zesíleném učení.

K důkazu neměnnosti znaků vůči rozměru tváře a natočení prvně spočteme podobnou transformaci k normalizaci aktuálního tvaru ke střednímu tvaru, která je odhadem pomocí metody nejmenších čtverců přes všechny význačné body obličeje. Dřívější práce např. [16] potřebují k transformaci obrazu současně vypočítané příslušné Haarovy příznaky. V tomto případě raději transformujeme pixelové souřadnice zpět do originálního obrazu ke spočtení rozdílu pixelů, což je mnohem efektivnější.

Jednoduchá cesta k indexaci pixelu je použít v základním tvaru globální souřadnice (x, y). To je dobré pro jednoduché tvary jako elipsy, ale nedostatečné pro poddajné tvary v obličeji. Protože vhodnější znaky jsou rozděleny do hlavních význačných bodů jako oči, nos a ústa (tzn. dobrý rozdíl znaku může být „střed oka je tmavší než nos“ nebo „dvě zorničky jsou podobné“) a lokace význačných bodů v obličeji se může měnit pro odlišné tváře.

V této práci je použito indexování pixelů pomocí lokálních souřadnic $(\delta x, \delta y)$ s ohledem na nejbližší význačný bod. Jak znázorňuje obr. 3 a) pixely indexované stejnými lokálními souřadnicemi mají významný smysl, ale b) pixely indexované globálními souřadnicemi mají významně rozdílné vnímání z důvodu možnosti natočení tváře.



Obr. 3: Indexace pixelů a) lokálně b) globálně [3]

V první fázi pro každý slabý regresor R^t náhodně vybereme P pixelů. Je vygenerováno P^2 rozdílových znaků. Nyní je novou výzvou jak rychle a efektivně vybrat znaky z tohoto velkého výběru možností.

4.3 Korelace výběru znaků

K tvaru dobrého fern regresoru F vybereme výstup z P^2 znaků. Obvykle se to uskuteční náhodným vygenerováním velkým množstvím fern a výběrem jednoho s minimální regresní chybou rovnice (10). Označme tuto metodu jako *n-Best*, kde n je počet vygenerovaných fern. z důvodu kombinatorické exploze je nemožné vyčíslit rovnici (10) pro všechny kompoziční znaky.

K lepšímu zkoumání rozsáhlého prostoru znaků v krátkém čase a generování dobrých kandidátů fern, použijeme korelaci mezi znaky a regresním cílem. Cíl je vektorový delta tvar, který je rozdílem mezi skutečným tvarem a aktuálním odhadem tvaru. Očekáváme, že dobrý fern bude splňovat dvě vlastnosti: 1) každý znak ve fern by měl být velmi rozdílný k regresnímu cíli, 2) korelace mezi znaky by měla být nízká tak, že při složení jsou komplementární.

[3] `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/cvpr12_facealignment.pdf`

K nalezení znaků splňující tyto vlastnosti navrhují autoři [2] korelačně-základní funkci výběru:

1. Návrh regresního cíle (vektorového delta tvaru) k náhodnému směru podle produktu skaláru.
2. Mezi P^2 znaků vybrat znak s nejlepší korelací ke skaláru.
3. Opakovat krok 1. a 2. dokud nezískáme F znaků.
4. Zkonstruovat fern z F znaků s náhodnou hranicí.

Náhodný průřez slouží dvěma účelům: může uchovat přibližnost takovou, že korelace znaků k projekci je diskriminační k delta tvaru; četné projekce mají nízkou korelaci s vysokou pravděpodobností a výběr znaků by měl být vhodně komplementární.

5 Local Binary Features

Pomocí algoritmu LBF můžeme efektivně a velmi přesně porovnávat obličeje. Tento přístup má dvě nové komponenty - sadu lokálních binárních znaků a lokální přístup při učení těchto znaků. Podstatou lokálního učení znaků je učení velmi charakteristických znaků nezávisle pro každý význačný bod v obličeji. Získané lokální binární znaky použijeme ke společnému učení lineární regresí. Tento přístup podle [1] dosahuje nejlepších výsledků v náročných testech. A protože je získání a regrese lokálních binárních znaků výpočetně velmi nenáročná, tento přístup je oproti jiným metodám mnohem rychlejší. Dosahuje přes 3 000 fps na počítačích nebo 300 fps na mobilních telefonech.

Charakteristický tvar regrese se ukazuje jako nejdůležitější přístup pro přesné a masivní porovnávání obličejů. To je především proto, že tyto přístupy mají některé odlišné vlastnosti: 1) jsou pouze charakteristické, 2) jsou schopny přizpůsobit výsledný tvar, 3) jsou schopny efektivně využít obsáhlé skupiny trénovacích dat.

Tvar regrese předpovídá tvar obličeje S v kaskádovém chování [2], [3]. Počáteční tvar S^0 postupně zpracujeme pomocí odhadu změny ΔS. Obecně můžeme vyjádřit tuto změnu jako:

$$\Delta S^t = W^t \Phi^t(I, S^{t-1}) \tag{14}$$

kde I je vstupní obraz, S^{t-1} je předchozí tvar regrese, Φ^t je mapovací funkce a W^t je matice lineární regrese.
Znaky naučené touto cestou jsou známé jako „shape-indexed“.

Vlastnost mapovací funkce Φ^t je zásadní při regresi tvarů. V předcházejících pracích tvar regrese určují buď ručně [6] nebo učením [2]. Ačkoli ruční označení a následné trénování funkce W^t fungovalo dobře, tak tento přístup není optimální pro specifické porovnávání obličejů. Proto použijeme přístup pomocí učení, který učí společně funkce Φ^t a W^t pomocí stromů s využitím random forests [7] na celém regionu, který je plný význačných bodů.

Znamená to, že přístup s pozdějším učení může být lepší, protože se díky tomu naučí specifické znaky. Jak je zmíněno v [6], tak lze tento postup použít pouze současně s ručním označením. Důvodem jsou velké možnosti funkce Φ^t. Uveďme si příklad. Použitím regionu bez ručního označení bude trénování na vstupu s velkým množstvím funkcí popisující znaky náročný a k tomu učení přes všechny kombinace těchto funkcí bude mít ve finále nereálnou výpočetní cenu. Druhým zásadním nedostatkem je, že mnoho těchto znaků by mohlo mít okolo mnoho šumu, což by mohlo způsobit falešně detekované znaky a poškodit následné učení.

V [1] se snaží ukázat lepší učící přístup. Princip lokálního učení je postaven na dvou pohledech: pro lokalizování spolehlivého bodu v každé fázi učení, 1) velmi charakteristická struktura informací leží v lokálním regionu poblíže odhadu význačných bodů z předchozí fáze, 2) lokace dalších bodů a lokální struktura tohoto bodu stanoví dostatečné informace pro správnou lokalizaci.

V [1] autoři navrhují následující dva typy regularizace pro učení Φ^t:

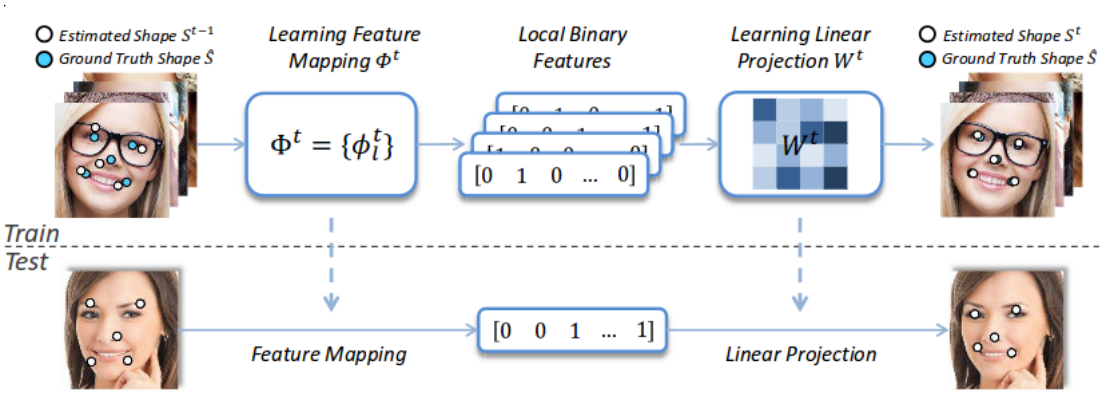
- Φ^t je složená z nezávislých lokálních znaků mapujících funkcí, tzn. $\Phi^t = [\Phi_1^t, \Phi_2^t, \ldots, \Phi_L^t]$ (L je počet význačných bodů)
- Každé Φ_l^t je naučeno pomocí nezávislé regrese l-tého význačného bodu v odpovídajícím lokálním regionu.

Touto regularizací efektivně eliminujeme většinu šumu nebo nevýznačných znaků. Redukce učení vede k lepšímu zobecnění.

Při učení každé Φ_l^t použijeme sadu regresních stromů k odvození binárních znaků. Po sloučení všech lokálních binárních znaků z tvaru mapující funkce Φ^t naučíme charakteristicky W^t pro globální odhad tvaru. Díky tomu najdeme takový dvou-fázový učící proces (určení lokálních binárních znaků a globální lineární regresi), který je mnohem lepší než spojení učení Φ^t a W^t pomocí stromové regrese jako v [2].

5.1 Regrese lokálních binárních znaků

V rovnici (14) je mapovací funkce znaků Φ^t a matice lineární regrese W^t neznámá. Jejich naučení provedeme v následujících dvou krocích: Prvně naučíme mapovací funkci znaků generovat lokální binární znaky pro každý význačný bod. Ty poté spojíme dohromady jako Φ^t. V druhém kroce vytvoříme matici W^t pomocí lineární regrese. Obr. 4 ukazuje přehled postupu.



Obr. 4: Přehled postupu tvorby Φ^t a W^t [4]

[4] http://home.ustc.edu.cn/~sqren/FaceAlignment/FaceAlignment_LocalBinaryFeatures.pdf

Jak lze vidět na Obr.4 v trénovací fázi začneme pomocí učení mapovací funkce znaků $\Phi^t(I, S^{t-1})$ generovat lokální binární znaky. Získáné znaky a odhad změny $\{\Delta\hat{\mathrm{S}}_i^t = \hat{\mathrm{S}}_i - S_i^{t-1}\}$ použijeme k naučení lineární projekce W^t pomocí lineární regrese. V testovací fázi je tvar změny přesně předpovězen a aplikován na aktuální odhadovaný tvar.

5.1.1 Učení funkce lokálních binárních znaků Φ^t

Mapující funkce znaků je složena ze sady lokálních znaků mapujících funkcí, tedy $\Phi^t = [\Phi_1^t, \Phi_2^t, \cdots, \Phi_L^t]$. Každou z nich učíme nezávisle na ostatních. Cíl regrese pro učení Φ_l^t je ručně označený tvar $\Delta\hat{\mathrm{S}}^t$:

$$\min_{\omega^t,\, \Phi_l^t} \sum_{i=1} ||\pi_l \circ \Delta\hat{\mathrm{S}}_i^t - \omega_l^t \Phi_l^t(I_i, S_i^{t-1})||_2^2 \tag{15}$$

kde i je iterace přes všechny trénovací příklady, operátor π_l extrahuje dva elementy $(2l-1, 2l)$ z vektoru $\Delta\hat{\mathrm{S}}_i$ a $\pi_l \circ \Delta\hat{\mathrm{S}}_i$ je ručně označený 2D-offset l-tého význačného bodu v i-tém trénovacím příkladě.

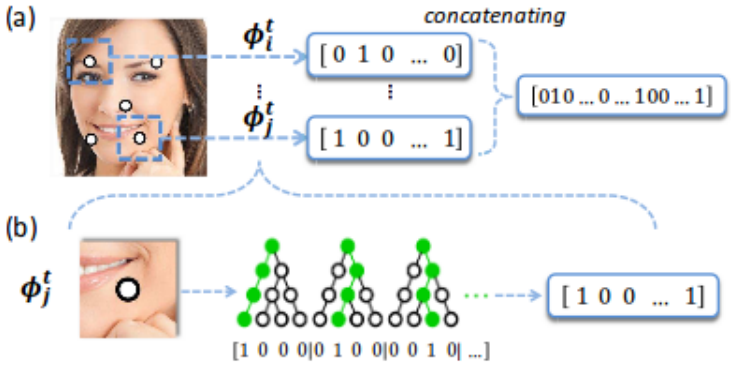
Použijeme standardní regresi RF k učení každé mapující funkce Φ^t. Oddělené uzly ve stromech jsou trénovány použitím rozdílu pixelů každého znaku [2]. K natrénovaní každého uzlu testujeme několik náhodných znaků a vybereme znak, který měl maximální odchylku při redukci. Po trénování uložíme listy uzlů do 2D-offset vektoru, který je průměrem všech trénovacích příkladů v daném listu.

Odhad lze vytvořit jenom pro pixel konkrétního znaku v lokálním regionu poblíž význačného bodu. Pro tento přístup je kritické použití těchto lokálních regionů. Během trénování se optimální velikost regionu odhaduje pomocí křížové validace (cross validation) pro každou fázi. Více informací najdete v [1], kapitole 3.3.

Výstupem random forest je suma výstupů uložených v každém listu pro každý uzel. Za předpokladu totálního čísla listů uzlů D, výstup může být popsán jako:

$$\omega_l^t \Phi_l^t(I_i, S_i^{t-1}) \tag{16}$$

kde ω_l^t je matice $2 \times D$, kde každý sloupec představuje 2D vektor pro odpovídající list uzlu a Φ_l^t je D-dimensionální binární vektor. Pro každou dimenzi ve Φ_l^t označíme hodnotou 1 pozice, kdy testovací příklad odpovídá listu v uzlu. Jinak na pozici vložíme hodnotu 0. Díky tomu bude Φ_l^t velmi řídký binární vektor. Počet nenulových prvků ve Φ_l^t je stejný jako počet stromů ve forest, což je mnohem menší než D. Toto nazveme „*local binary features*“. Obr. 5 ilustruje proces extrakce lokálních binárních znaků - local binary features.



Obr. 5: Local Binary Features [5]

5.2 Učení W^t pomocí globální lineární regrese

Po proběhnutí lokálního učení s využitím random forest získáme kromě binární funkce Φ_l^t také výstup lokální regrese ω_l^t. Tento výstup regrese nicméně vyřadíme. Raději spojíme binární znaky s globální mapující funkcí Φ^t a naučíme ji globální lineární projekci W^t pomocí minimalizace následující funkce:

$$W^t = \min_{W^t} \sum_{i=1}^{N} \parallel \Delta \hat{S}_i^t - W^t \Phi^t(I_i, S_i^{t-1}) \parallel_2^2 + \lambda \parallel W^t \parallel_2^2 \tag{17}$$

kde první část je regresní cíl, druhá část je L2 regularizace W^t a λ kontroluje regularizační sílu. Regularizace je nutná protože dimenze znaků je velmi vysoká. Pro experimenty s 68 význačnými body může být dimenze Φ^t rovna více než 100 000. Bez použití regularizace se dá pozorovat zbytečné přeučení. Protože binární znaky jsou vysoce řídké, použijeme dvojité souřadnice původní metody [8] ke spojení s velmi rozsáhlým řídkým lineárním systémem. Před funkcí s kvadratickou rovnicí s ohledem na W^t můžeme vždy dosáhnout globálního optima.

Najdeme takové globální „přeučení“ nebo „přenosové učení“, které významně zlepší výkon. Podle [1] se tak stane ze dvou důvodů. Za prvé lokálně naučený výstup pomocí random forest může obsahovat šum, protože počet trénovacích příkladů v listech může být nedostačující. Za druhé globální regrese může efektivně provést globální tvar omezení a zredukovat lokální chyby způsobené zakrytím a nejednoznačným vzhledem.

[5] http://home.ustc.edu.cn/~sqren/FaceAlignment/FaceAlignment_LocalBinaryFeatures.pdf

5.3 Princip lokalit

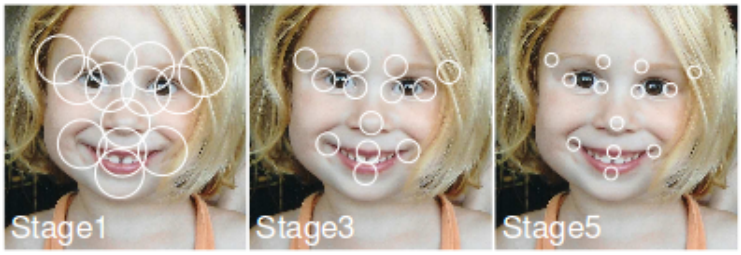
Jak již bylo několikrát řečeno, aplikujeme dvě důležité regularizační metody při učení znaků, kde používáme princip lokalit: 1) naučíme forest nezávisle pro každý význačný bod, 2) bereme ohled pouze na pixely znaků v daném lokálním regionu pro význačný bod. Podle [1] byly tyto volby z následujících důvodů.

Lokální region

Chceme předpovědět odchylku Δs z jednotlivých význačných bodů a chceme vybrat znaky z lokálních regionů s poloměrem r. Optimální poloměr r by měl záviset na rozložení Δs. Pokud je Δs pro všechny trénovací příklady hodně rozptýlené, budeme muset použít velký poloměr r, jinak bude poloměr mnohem menší.

Při studiu vztahu mezi rozložením Δs a optimálním poloměrem r pro každý význačný bod spojíme trénování a test vzorových regionů, jejichž Δs pocházejí z Gaussiánového rozdělení s různými standardními odchylkami. Pro každé rozložení experimentálně určíme optimální poloměr regionů (závisí na výsledcích testování chyby) pomocí trénování regresivních forest na různých poloměrech. Použijeme stejné parametry pro forest (hloubku stromů a počet stromů) jako při trénování. Tento experiment opakujeme pro každý význačný bod a zvolíme průměr hodnot pro optimální poloměr regionu.

V kaskádovém trénování v každé etapě hledáme nejlepší poloměr regionu (z 10-ti diskrétních hodnot) pomocí křížové validace [10] na validačním setu. Obr. 6 znázorňuje nejlepší poloměr regionů nalezených ve fázi 1, 3 a 5. Podle očekávání se poloměr postupně zmenšuje.



Obr. 6: Nejlepší poloměry lokálních regionů ve fázi 1, 3 a 5. [6]

[6] `http://home.ustc.edu.cn/~sqren/FaceAlignment/FaceAlignment_LocalBinaryFeatures.pdf`

Regrese jednotlivých význačných bodů v obličeji

Je potřeba ukázat, že nezávislá regrese pro každý význačný bod je nejoptimálnější. Například bychom pravděpodobně mohli minout dobrý znak tak, že může být sdílen pomocí rozmnožení význačných znaků. Nicméně, lokální regrese má mnohem méně výhod než globální učení použité v [2].

Za prvé sada funkcí v lokálním učení obsahuje méně šumu. Zde by mohlo být vhodnější použít funkce v globálním učení. Ale hodnota šumu v globálním učení může být nízká, což způsobí výběr správného znaku složitější.

Za druhé, použitím lokálního učení netvrdíme, že použijeme i lokální předpověď. V tomto přístupu využívá lineární regrese ve druhém kroce všech naučených lokálních znaků k vytvoření globálního odhadu. Protože lokální učení význačných znaků je nezávislé, výsledky znaků jsou přirozeně více odlišné a doplňují se každou další fází.

Nakonec zmíníme, že lokální učení je přizpůsobivé v různých krocích. V prvních fázích je lokální region relativně velký a kryje více význačných bodů. Funkce naučené z jednoho význačného znaku můžou opravdu pomoci se sousedními význačnými body. V pozdější fázi je region malý a lokální regrese doladí každý význačný bod. Lokální učení je tedy výhodnější pro pozdější fáze.

6 Pomocné funkce

Jak již bylo zmíněno, v této práci je využito funkce random forest, která je v této práci implementována.
Dále se používají geometrické transformace, konkrétně projektivní transformace. Dvě funkce (dopřednou afinní transformaci a získání transformační matice) bylo nutné si naprogramovat, protože žádná z dostupných knihoven nenabízela správnou funkci.

6.1 Random forest

Pojem random forest pochází z Random decision forests [23]. Principem je vytvoření skupiny stromů, která rozhoduje o zařazení objektu do daných tříd, u které je třeba vhodně zkombinovat klasifikační funkce jednotlivých stromů.
Nejznámější metody pro generování random forests jsou [7] nebo také tzv. bagging [24]. Random forests nezáleží příliš na kvalitě jednotlivých stromů. Cílem je minimalizovat chybu celého forest.

V této práci je využito random forest, který je přizpůsoben požadavkům algoritmu LBF. Více podrobností nalezneme v kapitole 7.2.

6.2 Afinní transformace

Afinní transformace je zobrazení bodů jednoho afinního prostoru do jiného afinního prostoru. Speciálním případem je tzv. afinita, nebo-li zobrazení afinního prostoru do téhož afinního prostoru. Afinní transformace jsou založeny na zvětšení, posunutí, otočení a zkosení.
Matematicky lze afinní transformaci vyjádřit jako $y = Ax + b$, kde x je bod z prvního afinního prostoru, A je transformační matice, b je parametr určující posunutí a y je afinním obrazem bodu x, tedy transformovaný bod.

Projektivní prostor

V grafickém zpracování obrazu můžeme při afinních prostorech narazit na jisté problémy (např. práce s body v nekonečnu). Proto je nutné zavést projektivní prostor. Podrobněji se o této problematice můžete dočíst například v [12].
Pro správnou funkčnost později použitého algoritmu dopředné afinní transformace je nutné zavést tzv. homogenní souřadnice. V praxi to znamená, že pro každý dvourozměrný bod X změníme souřadnice (x, y) na příslušné homogenní souřadnice (wx, wy, w), kde $w \in \mathbb{R} \setminus \{0\}$. Díky této úpravě můžeme nyní využít vztahu pro afinní transformace s využitím transformační matice z trojdimenzionálního prostoru.

Projektivní transformace

Vzhledem k využití homogenních souřadnic je transformace jednodušší, protože nám postačí pouze transformační matice. Nepotřebujeme již využívat vektor posunu b viz. Afinní transformace.

Příklady transformačních matic

Změna měřítka

$$\begin{bmatrix} z & 0 & 0 \\ 0 & z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Posunutí

$$\begin{bmatrix} 1 & 0 & b_x \\ 0 & 1 & b_y \\ 0 & 0 & 1 \end{bmatrix}$$

Otočení

$$\begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Zkosení

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Dopředná afinní transformace

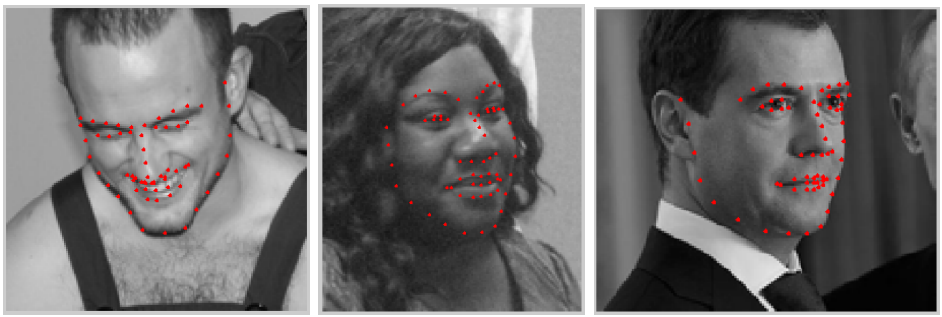
V této práci je využito dopředné afinní transformace. Transformační matice A je rozměru 3×3, ale body $X_0, X_1, \ldots, X_n$ jsou z dvoudimenzionálního souřadného systému. Proto využijeme již zmíněného přechodu do homogenních souřadnic, abychom mohli využít jednoduššího výpočtu transformace. Ukázku algoritmu najdete v kapitole Pomocné funkce.

Poznámka: Poslední řádek matice A musí vždy obsahovat nulové prvky, kromě prvku diagonálního, který bude mít hodnotu 1.

7 Popis implementace

7.1 Příprava na zpracování

Před začátkem samotné implementace algoritmů je nutné si předem připravit trénovací data. Jako tréninkový dataset jsem zvolila dataset AFW[7]. Použitý dataset obsahuje kromě testovacích obrázků také příslušné `*.pts` soubory. Tyto soubory obsahují 68 význačných bodů v obličeji (Obr. 1, kapitola 2.1), podle kterých se bude učící algoritmus trénovat.
Je tedy nutné si vždy načíst testovací obrázek a k němu příslušný `*.pts` soubor. Pomocí `*.pts` souborů se detekuje obličej, jak bylo popsáno v kapitole 2.1. Detekci obličeje a ruční označení význačných bodů si můžeme prohlédnout na obr. 7.



Obr. 7: Ručně označené význačné body v lokalizovaném obličeji

Jakmile získáme základní data z trénovacího datasetu, tak z důvodu lepšího učení algoritmu ještě všechny obrázky zrcadlově otočíme. Tím získáme dvojnásobně velký trénovací dataset, což vede k většímu počtu trénovacích obrázků a tedy lepšímu výsledku učícího algoritmu.
Následně pomocí získaných dat z obrázků a základního rozložení bodů v obličeji, viz. Obr. 1, určíme střední odhad vzdálenosti příslušných bodů a také transformační matici.

Když máme připraveny potřebná data pro každý trénovací obrázek, můžeme pokračovat dále. Prvně je nutné natrénovat random forest (kapitola 7.2). Následně pomocí RF určíme, kde se nacházejí lokální binární znaky (kapitola 7.3). Po získání souřadnic lokálních binárních znaků můžeme provést globální regresi (kapitola 7.4), ve které probíhá naučení trénování a poté vykreslení odhadu umístění význačných bodů do obrázku.

Tento postup (trénování random forest, lokalizace binárních znaků a globální regrese) proběhne v několika fázích, kde počet fází je určen nastavením parametru `max_numstage` v souboru `config_tr.py`. V aktuálním nastavení programu, který je přiložen, je tento parametr roven 4, tedy celý postup je zopakován čtyřikrát, aby se dosáhlo dobrého určení pozice význačných bodů v obličeji.

[7] `http://ibug.doc.ic.ac.uk/resources/facial-point-annotations/`

Po každé fázi uložíme vygenerovaný random forest a matici globální lineární regrese W^t. Tyto dvě proměnné z každé fáze budeme potřebovat ještě později při testování.

7.2 Random forest

Random forest, který je implementován v této práci, má následující nastavení:

počet náhodných znaků	individuální nastavení podle tabulky 2
bagging overlap	0.4
radius	[0.4,0.3,0.2,0.15,0.12,0.10,0.08,0.06,0.06,0.05]
počet stromů	5
hloubka stromů	4

Tabulka 1: Tabulka nastavení RF

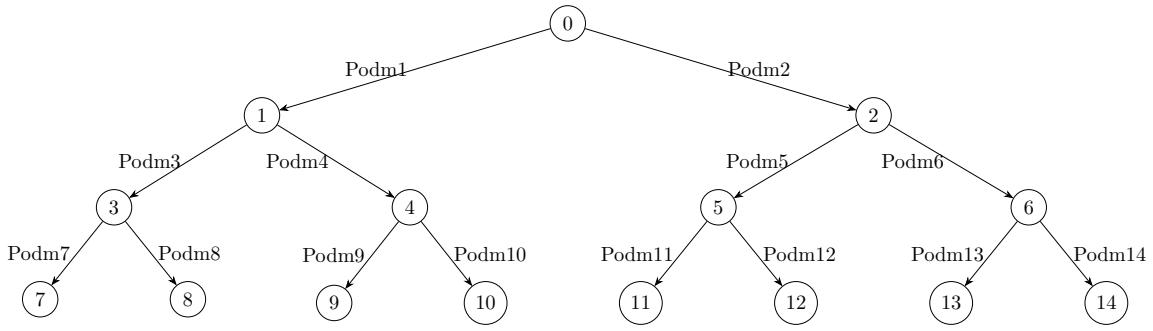
Po určení základních parametrů z tabulky 1 pokračujeme tím, že definujeme první průchod stromu, tedy nultý uzel. Nultý uzel obsahuje všechna čísla v rozmezí $(0, tmp)$, kde tmp je určeno následovně: $tmp = (e - s + 1) * params['augnumber']$. Hodnota s je určena jako $s = max((t-1) * Q - (t-1) * Q * overlap\_ratio + 1, 1)$, hodnota e je určena jako $e = min(s + Q, dbsize)$, kde t je aktuální strom, $Q = dbsize/((1 - overlap\_ratio) * (params['max\_numtrees']))$, $overlap\_ratio = params['bagging\_overlap']$, `dbsize` je počet obrázků se kterými pracujeme a `params` jsou předem definované parametry.

Nyní nastavíme všechny potřebné hodnoty nultého uzlu:

- `Ind_samples` $= (0, tmp)$
- `issplit` $= 0$
- `pnode` $= 0$
- `depth` $= 1$
- `cnodes` $=$ [0,0]
- `isleafnode` $= 0$
- `feat` $=$ [0,0,0,0]
- `thresh` $= 0$

`Ind_samples` jsou hodnoty daného uzlu, v nultém uzlu jsou uloženy všechny. `Issplit` je index určující, zda má uzel potomky. `Pnode` je číslo uzlu, `depth` aktuální hloubka stromu, `cnodes` jsou indexy potomků (pokud žádné neexistují, zůstává hodnota [0,0]), `isleafnode` určuje zda je daný uzel listem (0, pokud není list; 1 pokud je list) a hodnota `thresh` je vygenerované číslo podle které se poté dělí hodnoty v uzlu. Hodnota `feat` obsahuje čtyři čísla, první dvě určují průměrný úhel, poslední dvě průměrný poloměr. Pomocí hodnoty `feat` jsme schopni natočit masku obsahující binární znaky.

Jakmile máme nastavený nultý uzel, tak můžeme pokračovat s konstrukcí stromu. V poslední fázi získáme následující strom:



Obr. 8: Ukázka stromu z random forest

Popisky zapsané u hran stromu jsou podmínky, podle kterých se dělí dále hodnoty v daném uzlu. Podle podmínky se může stát, že v uzlu nebude žádný prvek, protože nebude vyhovovat podmínce. Pokud tato možnost nastane, tak do daného uzlu uložíme prázdné listy a při případném dělení uložíme do potomků takového uzlu také prázdné listy.

Počet stromů ve forest určuje parametr `max_numtrees`. V této práci byla hodnota počtu stromů ve forest rovna 5. Při vyšším počtu stromů nebo větší hloubce stromů by algoritmus vykazoval lepší výsledky, ale pro tuto implementaci bylo toto nastavení dostačující, což si můžeme prohlédnout v kapitole 8.

7.3 Lokální binární znaky

Jakmile máme k dispozici natrénovaný RF můžeme začít lokalizovat místa, kde se nacházejí význačné body. Na začátku je nutné spočítat dimenzi matice binárních znaků, abychom později mohli správně určit matici lineární regrese W^t.

Jedním z požadavků v této části implementace je spojit získané hodnoty `feat, isleafnode, thresh` a `cnodes` ze všech náhodných stromů do samostatných proměnných obsahující hodnoty ze všech náhodných stromů, které máme k dispozici. To lze provést například uvedeným kódem 1.

```
    id_rfnode = 0
    for t in range(params['max_numtrees']):
        feats[l][id_rfnode:(id_rfnode + rf[t]['num_nodes'] + 1), :] =
            rf[t]['feat'][0: rf[t]['num_nodes'] + 1, :].copy()
        isleaf[l][id_rfnode:(id_rfnode + rf[t]['num_nodes'] + 1), :] =
            rf[t]['isleafnode'][0: rf[t]['num_nodes'] + 1,:].copy()
        threshs[l][id_rfnode:(id_rfnode + rf[t]['num_nodes'] + 1), :] =
            rf[t]['thresh'][0: rf[t]['num_nodes'] + 1, :].copy()
        cnodes[l][id_rfnode:(id_rfnode + rf[t]['num_nodes'] + 1), :] =
            rf[t]['cnodes'][0: rf[t]['num_nodes'] + 1, :].copy()

        id_rfnode = id_rfnode + rf[t]['num_nodes'] + 1
```

Výpis 1: Spojení hodnot ze všech náhodných stromů

Následně se zpracují pixely obrázku a je nutné spočítat matici, ve které bude zaznačeno, zda se na dané pozici pixelu nachází význačný bod. Pokud se v daném místě význačný bod nachází, uložíme do matice hodnotu 1, jinak ponecháme hodnotu 0. Tvorba řádků této matice je prezentována kódem 2.

```
for t = 1:params.max_numtrees
    num_nodes~= (rf{t}.num_nodes);
    id_cnode = 1;
    while(1)
        if isleaf(id_cnode + cumnum_nodes)
            binfeature(cumnum_leafnodes~+ find(rf{t}.id_leafnodes~== id_cnode)) = 1;
            cumnum_nodes~= cumnum_nodes~+ num_nodes;
            cumnum_leafnodes~= cumnum_leafnodes~+ rf{t}.num_leafnodes;
            break;
        end
        id_cnode = ind_cnodes(cumnum_nodes~+ id_cnode, cind(cumnum_nodes~+ id_cnode));
    end
end
```

Výpis 2: Lokalizace binárních znaků

Výsledkem této funkce je řídká matice obsahující jedničky pouze na pozicích, kde byl lokalizován význačný bod. Díky této matici můžeme algoritmus dále trénovat a získat zmíněnou matici lineární regrese W^t.

7.4 Globální regrese

Globální regrese využívá knihovny liblinear [8] k natrénování matice lineární regrese W^t. Tato knihovna je volně ke stažení a pokud chceme otestovat implementaci v Pythonu je nutné ji nainstalovat. Tato knihovna je zásadní pro správnou funkčnost algoritmu.

Abychom mohli spustit trénování a předešli zbytečnému přeučení funkce, je nutné ji nastavit parametry se kterými má být spuštěna. Pro tuto implementaci bylo zvoleno nastavení `param = '-s 12 -p 0 -c 0 -q'.format(cValue)`, kde `cValue = 1.0/binaryfeatures.shape[0]`. Parametr s určuje typ řešiče, který použijeme, pod řešičem číslo 12 se skrývá L2-regularizace. Parametr p nastavuje ztrátovost funkce, parametr c cenu a parametr q zajistí mód bez výpisu. Více o nastavení parametrů si můžete přečíst v dokumentaci ke knihovně liblinear [8].

Kód 3 ukazuje volání funkce train z knihovny `liblinear` a také způsob jak v Pythonu získat příslušné hodnoty do matice W^t.

```
1  #ll je naimportovana knihovna liblinearutil, ktera je soucasti sady knihoven liblinear.
2  for o~in range(deltashapes.shape[1]):
3      #Poznamka: upraven liblinear - issparse=False!!
4      model = ll.train(deltashapes[:,o].tolist(), binaryfeatures.tolist(), param)
5      for i in range(binaryfeatures.shape[1]):
6          W_liblinear[i,o] = model.w.__getitem__(i)
7
8  #Ziskani matice globalni linearni regrese
9  W = W_liblinear.copy()
```

Výpis 3: Trénovací funkce pro získání matice lineární regrese W^t

Po získání této matice zbývá dopočítat odhad umístění význačných bodů. K tomu využijeme pomocné funkce dopředné afinní transformace (kapitola 7.6). Po určení odhadu těchto význačných znaků je algoritmus téměř u konce. Všechny zmíněné funkce se opakují v několika fázích, jejichž počet určuje parametr `max_numstage`, který byl během odzkoušení této implementace nastaven na hodnotu 4.

v momentě, kdy získáme všechna potřebná data, tedy ruční označení, odhad lokace bodů a předchozí či počáteční umístění význačných bodů je můžeme také vykreslit. Na vykreslení je použita knihovna `matplotlib.pyplot` a kód pro vykreslení si můžete prohlédnout níže.

```
1  def drawshapes(image, shapes):
2
3      plt.interactive(False)
4      plt.figure()
5      plt.imshow(image, cmap = plt.get_cmap('gray'))
6
7      shape_gt = shapes[0]
8      shape_stage = shapes[1]
9      shape_newstage = shapes[2]
10
11     plt.scatter(x=shape_gt[:,0].tolist(), y=shape_gt[:,1].tolist(),c = 'r', s=20)
12     plt.scatter(x=shape_stage[:,0].tolist(), y=shape_stage[:,1].tolist(),c = 'y', s=20)
13     plt.scatter(x=shape_newstage[:,0].tolist(), y=shape_newstage[:,1].tolist(),c = 'b', s=20)
14
15     try:
16         drawshapes.counter += 1
17     except AttributeError:
18         drawshapes.counter = 0
19     #plt.show(block=True)
20     plt.savefig('obr{0}.png'.format(drawshapes.counter))
21     #raw_input('Pokracujte stiskem klavesy Enter ')
```

Výpis 4: Kód pro vizuální zobrazení lokace význačných bodů

V základním nastavení se uloží postupně obrázky s vyznačenými odhady lokace význačných bodů. Pokud bychom potřebovali tyto obrázky zobrazit a vykreslit, stačí odkomentovat řádky č. 19 a 21, a zároveň zakomentovat řádek č. 20.

7.5 Globální predikce

Globální predikce neboli předpověď, kde se budou význačné body nacházet, se využívá v případě, že máme již natrénovanou matici lineární regrese W^t a random forest. Poté tato funkce pracuje na velmi podobném principu jako funkce globální regrese, pouze s tím rozdílem, že nyní se již nevyužívá knihovny `liblinear` k trénování, protože danou matici W^t máme již k dispozici.

Stejně jako u globální regrese se zde postupně zpřesňuje odhad lokace význačných bodů a následně se může vykreslit.

7.6 Pomocné funkce

7.6.1 Dopředná afinní transformace

V kapitole 6.2 jsme představili algoritmus dopředné afinní transformace. Abychom jej mohli implementovat, je nutné prve pochopit, jak tento algoritmus funguje.

Využijme k tomu následující příklad:
Mějme transformační matici A, vektor x, který bude obsahovat veškeré x-ové souřadnice všech bodů a vektor y, který bude obsahovat veškeré y-ové souřadnice všech bodů.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad x = (x_0, x_1, \ldots, x_n), \quad y = (y_0, y_1, \ldots, y_n)$$

Algoritmus si automaticky doplní dvourozměrné souřadnice na příslušné homogenní souřadnice, kde w=1. Poté přepočítá souřadnice následujícím způsobem:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \ldots & x_n \\ y_1 & y_2 & \ldots & y_n \\ 1 & 1 & \ldots & 1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & \ldots & u_n \\ v_1 & v_2 & \ldots & v_n \\ 1 & 1 & \ldots & 1 \end{bmatrix},$$

Vzhledem k tomu, že jsme matici A zvolili tak, aby poslední řádek byl vyjma diagonálního jednotkového prvku nulový, tak je zpětný převod do afinních souřadnic jednoduchý - pouze zanedbáme třetí rozměr. Díky tomu jsou tedy (u_i, v_i) příslušné transformované souřadnice bodu (x_i, y_i), kde $i = 0, \ldots, n$. Souřadnice transformovaného bodu X_i získáme následovně:

$u_i = a_{11}x_i + a_{21}y_i + a_{31}$
$v_i = a_{12}x_i + a_{22}y_i + a_{32}$

K praktické implementaci v Pythonu využijeme knihovny Numpy. Algoritmus je znázorňen níže:

```
def transformPointsForward(T,v1,v2):
    if v1.shape[1] != 1 or v2.shape[1] != 1:
        print('Vektory musi byt sloupcove!')
        return
    elif v1.shape[0] != v2.shape[0]:
        print('Vektory musi byt stejne dlouhe!')
        return

    vecSize = v1.shape[0]
    concVec = np.concatenate((v1,v2),axis=1)
    onesVec = np.ones((vecSize,1))
    u~= np.concatenate((concVec,onesVec),axis=1)
    retMat = np.dot(U,T[:,0:2])

    return (retMat[:,0].reshape((vecSize,1)), retMat[:,1].reshape((vecSize,1)))
```

Výpis 5: Dopředná afinní transformace v Pythonu

8 Experimenty

Veškeré experimenty byly provedeny na notebooku obsahující dvoujádrový procesor Intel Core i5-5200u a RAM velikosti 8GB. Pracovním prostředím byl operační systém Ubuntu 16.04 LTS, prostředí pro programování PyCharm a knihovna openCv verze 3.1.0.

8.1 Nastavení parametrů

Nastavení parametrů pro otestování funkčnosti algoritmu LBF probíhalo ve třech různých nastaveních. U každého nastavení bylo použito datasetu AFW[8], který obsahuje 337 fotografií. Pro lepší naučení algoritmu byly tyto fotografie použity v originálním stavu a poté ještě zrcadlově otočeny. Celkově se tedy algoritmus učil na 674 fotografiích.

Celkově je detekce význačných částí provedena ve 4 krocích. Vyšší počet kroků funkce ať při trénování nebo testování samozřejmě vede k lepším výsledkům, nicméně již označení význačných tvarů po čtvrtém kroku mělo velmi dobré výsledky, proto jsem z důvodu časové náročnosti upustila od vyššího počtu kroků.

Nastavení RF bylo následovné:

- Bagging overlap = 0.4
- Poloměr lokálních regionů = [0.4,0.3,0.2,0.15,0.12,0.10,0.08,0.06,0.06,0.05]
- Počet stromů = 5
- Hloubka stromů = 4
- Max. počet thresholds = 500

Nastavení maximálního počtu náhodně generovaných znaků (pole hodnot):

Nastavení č.1	[10, 10, 10, 5, 5, 5, 4, 4]
Nastavení č.2	[50, 50, 50, 25, 25, 25, 20, 20]
Nastavení č.3	[100, 100, 100, 50, 50, 50, 40, 40]

Tabulka 2: Tabulka nastavení RF - parametr pro počet náhodně generovaných znaků

Jak si lze všimnout v tabulce č.2, celkem jsem testovala algoritmus ve třech různých nastavení. Nyní zhodnotím získané výsledky pro tyto nastavení v trénovací fázi algoritmu.

[8] http://ibug.doc.ic.ac.uk/resources/facial-point-annotations/

8.2 Porovnání získaných výsledků trénování s výsledky z MatLabu

Autoři práce [1] zvěřejnili implementaci v MatLabu, která je volně dostupná [5]. Díky přístupu k této implementaci můžu porovnat dosažené výsledky.

Začla jsem porovnáním časové náročnosti algoritmu. V níže uvedených tabulkách jsou zapsané získané časy, jak z mojí implementace v Pythonu, tak implementace v MatLabu, která jak již bylo zmíněno, je volně ke stažení.

Nejprve jsem otestovala nastavení č.1 (tabulka 2). V tabulce 3 jsou uvedeny výsledky trénování v Pythonu, v tabulce 4 jsou uvedeny výsledky trénování v MatLabu.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	1096.3354	1207.1477	1164.0509	1063.5184
Binární znaky	50.4075	47.2246	49.3976	45.8121
Trénování	63.6366	62.2024	60.7737	59.5192
Globalní regrese	72.5636	73.7370	71.4575	71.1425

Tabulka 3: Trénování č.1 - časová náročnost jednotlivých funkcí v Pythonu (v sekundách)

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	82.1509	77.5782	77.0730	76.1080
Binární znaky	18.9991	17.0186	16.8905	17.3581
Trénování	2.4962	2.1300	2.0995	2.5713
Globalní regrese	4.1638	3.5748	3.5535	4.1531

Tabulka 4: Trénování č.1 - časová náročnost jednotlivých funkcí v MatLabu (v sekundách)

Jak lze pozorovat, tak se časová náročnost jednotlivých implementací velmi liší. Zatímco u původní implementace v MatLabu trvá jeden průchod zhruba 2 minuty, tak u mé implementace v Pythonu trvá jeden průchod zhruba 22 minut. Má implementace je tedy výrazně časově horší.

Toto může být způsobeno například použitými datovými typy, kdy u mé implementace je nutné mít uložené velké množství dat ve slovníku nebo listu. Projít v Pythonu list slovníků, který obsahuje 674 slovníků je časově náročnější než v MatLabu projít strukturu.

Dále je možné pozorovat, že v rámci jednoho nastavení je časová náročnost jednotlivých průchodů zhruba stejná.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	1075.7220	1106.6005	1513.0143	1099.5467
Binární znaky	45.1413	46.5812	64.1568	45.0513
Trénování	58.9984	65.9794	94.8515	58.6231
Globalní regrese	68.1675	77.0902	112.3335	69.5808

Tabulka 5: Trénování č.2 - časová náročnost jednotlivých funkcí v Pythonu (v sekundách)

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	88.4858	87.1755	86.8099	81.4562
Binární znaky	17.0631	17.0077	16.9446	16.9712
Trénování	2.1037	2.1547	2.0854	2.0943
Globalní regrese	3.5868	3.6093	3.5660	3.5565

Tabulka 6: Trénování č.2 - časová náročnost jednotlivých funkcí v MatLabu (v sekundách)

Srovnatelné výsledky jsem získala z nastavení č.2, z tabulek č. 5 a 6, takže si lze všimnout, že se získané časy obou implementací opět značně liší, ale v rámci zpřesňování odhadu zůstávají přibližně stejné.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	1092.6962	1186.1468	1152.2947	1097.3384
Binární znaky	46.2138	46.3329	46.6230	46.0835
Trénování	59.2197	60.0629	58.6655	59.3605
Globalní regrese	68.2216	69.2189	69.6591	70.7925

Tabulka 7: Trénování č.3 - časová náročnost jednotlivých funkcí v Pythonu (v sekundách)

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Random forest	97.3211	97.5614	97.7260	86.7513
Binární znaky	16.9486	16.8833	16.9392	19.4211
Trénování	2.2516	2.1635	2.0999	2.8111
Globalní regrese	3.7725	3.6310	3.5638	4.8657

Tabulka 8: Trénování č.3 - časová náročnost jednotlivých funkcí v MatLabu (v sekundách)

I třetí nastavení má srovnatelné výsledky s předchozími volbami parametrů. S ohledem na tyto výsledky jsem došla k závěru, že nastavení parametru pro počet náhodně generovaných znaků algoritmus může vylepšit, nicméně časovou náročnost výrazně neovlivní.

Po otestování časové náročnosti jsem se rozhodla ještě porovnat tzv. Mean Square Error. Jedná se o chybu určující o kolik procent se liší aktuální odhad význačných tvarů od skutečného tvaru. V tabulkách 9 a 10 vidíme, že při každém ze tří použitých nastavení se v MatLabu i Pythonu chyba snižuje, tedy odhad se výrazně zlepšuje.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Trénování č.1	13.0519	6.9104	4.2345	2.8114
Trénování č.2	13.5316	7.3124	4.5373	3.0396
Trénování č.3	13.9043	7.6307	4.8278	3.2678

Tabulka 9: Mean Square Error při trénování v Pythonu (v procentech)

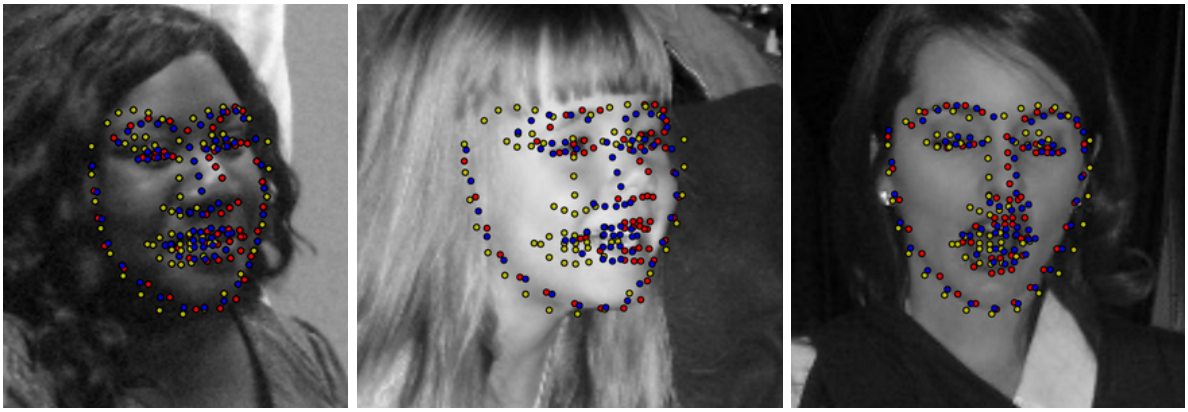
	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Trénování č.1	10.2410	4.6757	2.7437	1.7618
Trénování č.2	9.9285	4.5028	2.6174	1.7067
Trénování č.3	9.6383	4.3849	2.6045	1.7124

Tabulka 10: Mean Square Error při trénování v MatLabu (v procentech)

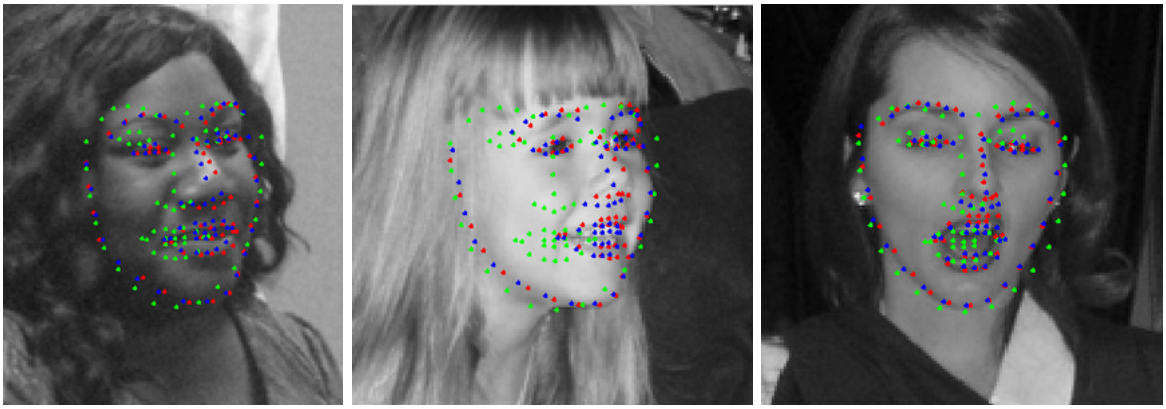
V tabulkách lze také pozorovat, že implementace v Matlabu je lehce lepší, co se určení odhadu týče. Toto může být způsobeno například tím, že Python a MatLab načítají hodnoty v obraze trochu jinak, což může slabě ovlivnit určení správného význačného bodu.

Další možností, která může ovlivnit výsledek, je funkce `train` z knihovny `liblinear`. Když jsem testovala funkčnost této knihovny, konkrétně tedy funkce train, tak jsem si všimla, že se získané výsledky liší v řádech desetin, místy jednotek. Díky tomu se může lehce zkreslit výsledná hodnota a následně posunout souřadnice pixelu, kde by se měl nacházet význačný bod v obličeji, což má za následek trochu horší odhad.

Nyní můžeme ještě porovnat vizuálně výsledky algoritmů. Vizuální porovnání je provedeno pouze pro nastavení č.2, tedy pro počet náhodně generovaných znaků s hodnotami v poli: $[50, 50, 50, 25, 25, 25, 20, 20]$.



Obr. 9: Vizuální výsledky implementace v Pythonu, nastavení č.2, první průchod



Obr. 10: Vizuální výsledky implementace v MatLabu, nastavení č.2, první průchod

V Pythonu je maska význačných bodů zelenožlutá, modrou barvou je označen aktuální odhad pozice význačných bodů a červeně je znázorněno ruční označení význačných bodů. V MatLabu je červeně také ruční odhad, modře aktuání odhad a zeleně maska.

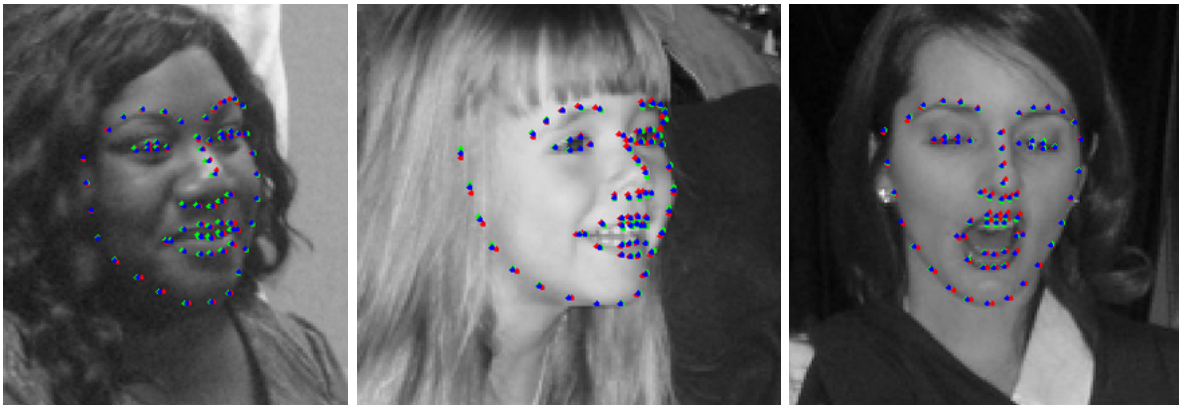
I ve vizuálním porovnání si lze všimnout, že implementace v MatLabu má v prvním průchodu algoritmu přesnější odhad než implementace v Pythonu. Ukážeme si i druhý, třetí a čtvrtý průchod.

Obr. 11: Vizuální výsledky implementace v Pythonu, nastavení č.2, druhý průchod

Obr. 12: Vizuální výsledky implementace v MatLabu, nastavení č.2, druhý průchod

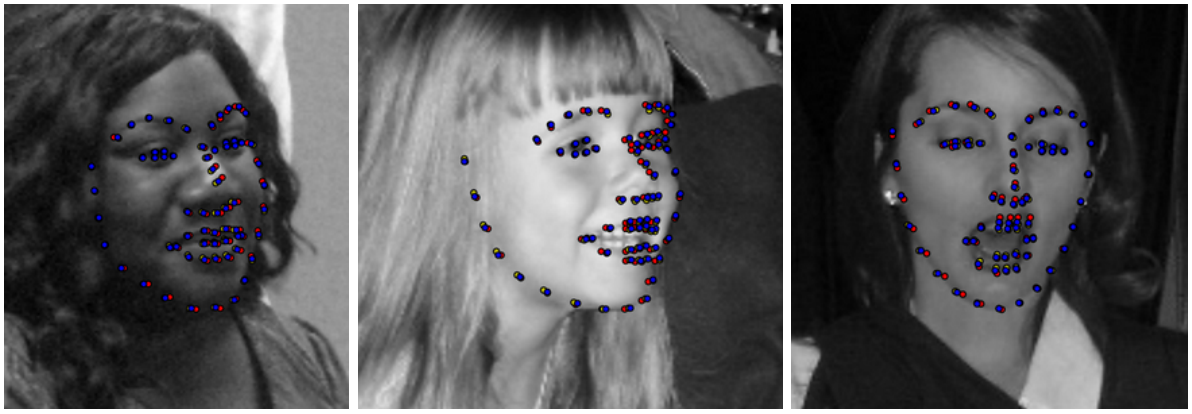
Ve druhém průchodu (obrázky 11 a 12) je možnost stále pozorovat, že algoritmus v MatLabu pracuje o trošku lépe než algoritmus v Pythonu. Jak již bylo zmíněno, může to být způsobeno například tím, že Python má přesnější početní operace, tudíž výsledky nejsou tolik zaokrouhleny.

Obr. 13: Vizuální výsledky implementace v Pythonu, nastavení č.2, třetí průchod

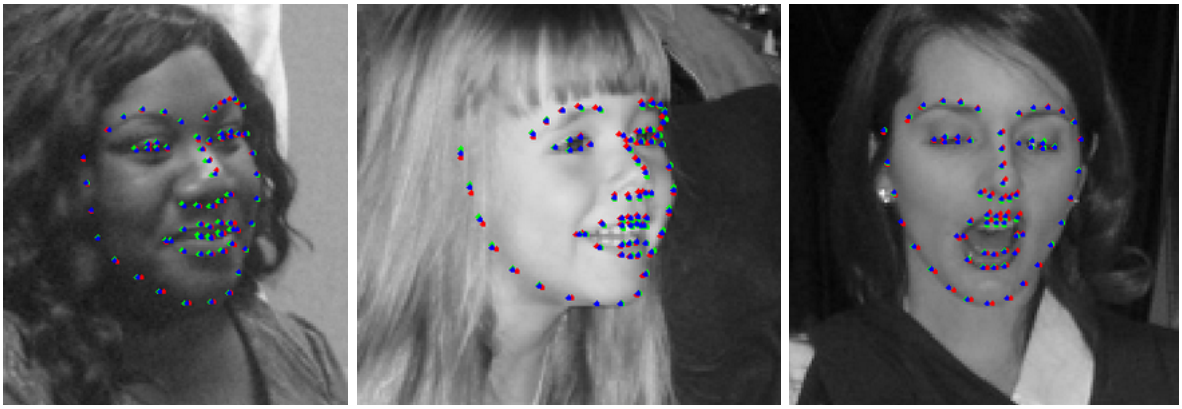


Obr. 14: Vizuální výsledky implementace v MatLabu, nastavení č.2, třetí průchod

Ve třetím průchodu se vizuální rozdíl téměř eliminoval. Vykreslení lokací význačních bodů vypadá pro lidské oko stejně, ačkoli podle tabulek 9 a 10 vím, že se Mean Square Error liší a přesnější je implementace v MatLabu.



Obr. 15: Vizuální výsledky implementace v Pythonu, nastavení č.2, čtvrtý průchod



Obr. 16: Vizuální výsledky implementace v MatLabu, nastavení č.2, čtvrtý průchod

V poslední fázi průchodu je vizuální rozdíl lokace význačných bodů téměř neviditelný. Vizuálním zobrazením jsem se jenom přesvědčila, že algoritmus funguje správně a detekce význačných bodů se s každým průchodem zlepšuje.

8.3 Porovnání získaných výsledků testování s výsledky z MatLabu

Při druhé fázi experimentů jsem porovnávala výsledky testování. Nechala jsem nastavení č.2, tzn. počet náhodně generovaných znaků v poli: $[50, 50, 50, 25, 25, 25, 20, 20]$.

Opět jsem prvně porovnala časovou náročnost algoritmu v Pythonu a v MatLabu.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Binární znaky	48.5297	48.7986	48.9421	49.9953
Globální předpověď	1.0652	1.0664	0.9749	0.7832

Tabulka 11: Časová náročnost testování v Pythonu pro nastavení č.2 (v sekundách)

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
Binární znaky	18.8625	18.3535	18.5992	18.5703
Globální předpověď	1.6037	1.4958	1.4884	1.5087

Tabulka 12: Časová náročnost testování v MatLabu pro nastavení č.2 (v sekundách)

Z tabulek 11 a 12 je patrné, že se výrazněji liší čas pro detekci binárních znaků. V Pythonu trvá zhruba 50 sekund lokalizovat binární znaky, kdežto v MatLabu cca 20 sekund.

Čas trvání globální předpovědi je u Pythonu kratší. Vzhledem k tomu, že předpověď u obou implementací trvá 1-1,5 sekundy, je tento čas v rámci časové složitosti celé implementace zanedbatelný.

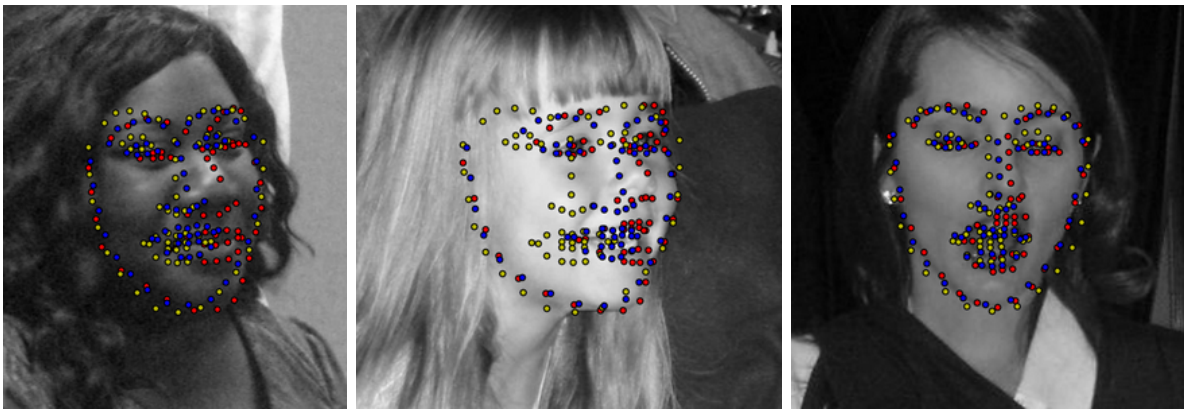
Podstatnější je chyba nazvaná Mean Square Error. Ta nám opět ukazuje o kolik se liší odhad určený algoritmem oproti skutečnému umístění význačných bodů.

	Průchod 1	Průchod 2	Průchod 3	Průchod 4
MatLab	12.1130	8.0459	7.1036	6.7624
Python	15.8967	11.1978	9.4221	8.7424

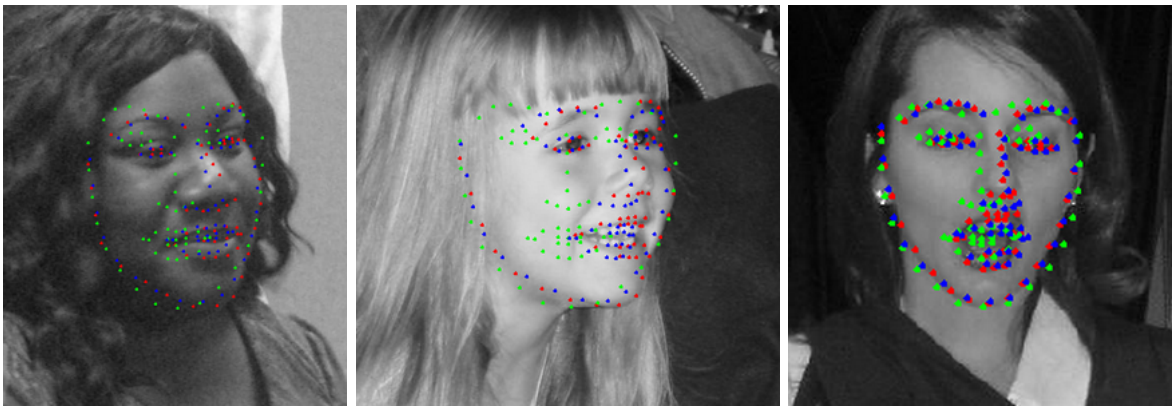
Tabulka 13: Mean Square Error při testování pro nastavení č.2 (v procentech)

V tabulce 13 vidíme, že implementace v MatLabu, tak i má implementace v Pythonu se s počtem průchodů zlepšuje v předpovědi umístění význačných bodů. Opět je zřetelně vidět, že původní implementace v MatLabu je lehce přesnější, než má implementace v Pythonu.

Také nyní můžu ukázat vizuální srovnání. Níže vidíte vizuální výsledky pro první průchod algoritmu pro testování.



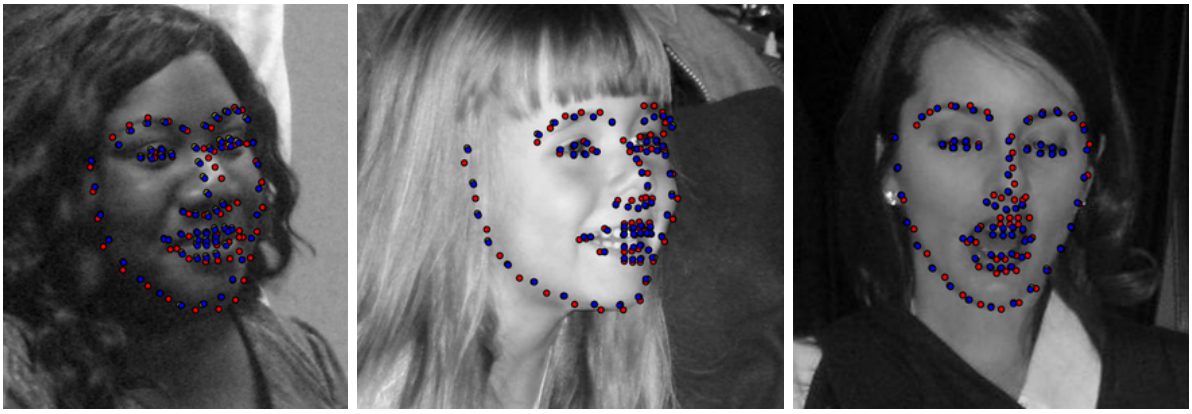
Obr. 17: Vizuální výsledky implementace v Pythonu, nastavení č.2, první průchod pro testování



Obr. 18: Vizuální výsledky implementace v MatLabu, nastavení č.2, první průchod pro testování

Jde vidět, že zelené (respektivě zelenožluté) zobrazení rysů je původní maska, modrá barva je aktuální odhad, který se přiblížil od počátečního umístění k ručnímu označení, které je znázorněno červeně.

Nyní přeskočím vizuální zobrazení druhého a třetího průchodu a rovnou si můžete prohlédnout poslední, tedy čtvrtý, průchod algoritmem a jeho vizuální výsledky.



Obr. 19: Vizuální výsledky implementace v Pythonu, nastavení č.2, čtvrtý průchod pro testování

Obr. 20: Vizuální výsledky implementace v MatLabu, nastavení č.2, čtvrtý průchod pro testování

Jak si lze všimnout i při testování se výsledné vizuální zobrazení lokace význačných bodů pro lidské oko jeví stejné pro obě implementace, ačkoli v tabulce 13 vidíme, že se hodnoty jednotlivých implementací liší.

9 Závěr

Výsledky experimentů ukazují, že implementace v Pythonu je oproti původní implementaci v MatLabu podstatně pomalejší. Velký rozdíl časové náročnosti implementace v Pythonu může být způsoben volbou datových typů, které v daném způsobu použití nemusely být nejvhodnější.

Vizuálně se výsledky téměř neliší a ačkoli se hodnoty Mean Square Error lehce lišily, tak získané výsledky hodnotím kladně. Algoritmus dokáže lokalizovat význačné body a během trénování se zpřesňuje.

Jedinou pozorovatelnou nevýhodu vidím v časové náročnosti, která je v Pythonu několikanásobně horší než v implementaci v MatLabu. Proto jsem došla k závěru, že implementace v Pythonu přiložená k této práci není vhodná pro reálné použití.

Zároveň se zjištěním, že algoritmus k funkčnosti potřebuje ruční označení význačných bodů, jsem došla k názoru, že není vhodný pro běžné použití detekce obličeje a význačných bodů. Pro využití v běžném životě doporučuji jiné algoritmy, které nevyžadují předem označené význačné body.

K dosažení lepších výsledků navrhuji vyzkoušet paralelizovat některé části, což by jistě vedlo ke zrychlení práce algoritmu. Dále by bylo vhodné se zamyslet nad použitými datovými typy, zda není možnost zvolit vhodnější datové typy a tím ušetřit na velikosti ukládaných dat.

Během práce na této diplomové práci jsem pronikla hlouběji do problémů analýzy obrazu. Získala jsem praktické zkušenosti ohledně zpracování obrazu a rozšířila své znalosti programování nejen v oblasti analýzy obrazu, ale také se naučila lépe programovat v Pythonu.

Literatura

[1] Shaoqing Ren, Xudong Cao, Yichen Wei, Jian Sun: Face Alignment at 3000 FPs via Regressing Local Binary Features, volně dostupné na http://home.ustc.edu.cn/~sqren/FaceAlignment/FaceAlignment_LocalBinaryFeatures.pdf

[2] Xudong Cao, Yichen Wei, Fang Wen, Jian Sun"Face Alignment by Explicit Shape Regression, volně dostupné na https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/cvpr12_facealignment.pdf

[3] Piotr Dollár, Peter Welinder, Pietro Perona: Cascaded Pose Regression, volně dostupné na https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/cvpr12_facealignment.pdf

[4] Zdroj použitého datasetu AFW dostupný na http://ibug.doc.ic.ac.uk/resources/facial-point-annotations/

[5] Implementace algoritmu LBF v MatLabu, volně dostupná na https://github.com/jwyang/face-alignment

[6] Xuehan Xiong, Fernando De La Torre: Supervised Descent Method and its Applications to Face Alignment, volně dostupné na http://www.cv-foundation.org/openaccess/content_cvpr_2013/papers/Xiong_Supervised_Descent_Method_2013_CVPR_paper.pdf

[7] Leo Breiman: Random Forests, volně dostupné na http://machinelearning202.pbworks.com/w/file/fetch/60606349/breiman_randomforests.pdf

[8] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang: LIBLINEAR: A Library for Large Linear Classi cation Chih-Jen Lin, volně dostupné na: http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf

[9] Ming-Hsuan Yang, David J. Kriegman, Narendra Ahuja: Detecting Faces in Images: A Survey, volně dostupné na http://faculty.ucmerced.edu/mhyang/papers/pami02a.pdf

[10] Jeff Schneider: Cross-Validation, volně dostupné na https://www.cs.cmu.edu/~schneide/tut5/node42.html

[11] M. Ozuysal, M. Calonder, V. Lepetit, P. Fua: Fast Keypoint Recognition using Random Ferns, volně dostupné na: http://cvlabwww.epfl.ch/publications/publications/2010/OzuysalCLF10.pdf

[12] Eduard Sojka, Martin Němec, Tomáš Fabián: Matematické základy počítačové grafiky, volně dostupné na http://mrl.cs.vsb.cz/people/sojka/pg/mzpg.pdf

[13] N. Duffy, D. P. Helmbold: Boosting methods for regression, volně dostupné na: https://users.soe.ucsc.edu/~dph/mypubs/RegressionBoostMLJ2002.pdf

[14] Iain Matthews, Simon Baker: Active Appearance Models Revisited, volně dostupné na: http://www.ri.cmu.edu/pub_files/pub4/matthews_iain_2004_2/matthews_iain_2004_2.pdf

[15] T.F.Cootes, C.J.Taylor: Active Shape Models, volně dostupné na: http://www.bmva.org/bmvc/1992/bmvc-92-028.pdf

[16] David Cristinacce, Tim Cootes: Boosted Regression Active Shape Models, volně dostupné na http://www.dcs.warwick.ac.uk/bmvc2007/proceedings/Papers/131/bmvc2007_pap_final.pdf

[17] P.Viola, M.J.Jones: Robust real-time face detection

[18] P.Felzenszwalb, D. Huttenlocher: Efficient matching of pictorial structures, volně dostupné na http://cs.brown.edu/~pff/papers/blobrec2.pdf

[19] C.Lampert, M.Blaschko, T.Hofmann, S.Zurich: Beyond sliding windows: Object localization by efficient sub-window search, volně dostupné na http://pub.ist.ac.at/~chl/papers/lampert-cvpr2008a-slides.pdf

[20] F.Fleuret, D.Geman: Coarse-to-fine face detection, volně dostupné na https://www.semanticscholar.org/paper/Coarse-to-Fine-Face-Detection-Fleuret-Geman/9b535f4edc4cbf8d4fb6182ec6b5c54db3c1cccb/pdf

[21] F.Fleuret, D.Geman: Stationary features and cat detection, volně dostupné na http://www.idiap.ch/~fleuret/papers/fleuret-geman-jmlr2008.pdf

[22] David Cristinacce, Tim Cootes: Boosted Regression Active Shape Models, volně dostupné na http://www.dcs.warwick.ac.uk/bmvc2007/proceedings/Papers/131/bmvc2007_pap_final.pdf

[23] Tin Kam Ho: Random decision forests

[24] Leo Breiman: Bagging Predictors, volně dostupné na http://www.cs.utsa.edu/~bylander/cs6243/breiman96bagging.pdf

[25] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011)

[26] Paul Viola, Michael Jones: Rapid Object Detection using a Boosted Cascade of Simple Features, volně dostupné na https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf

A Příloha: Obsah přiloženého CD

CD obsahuje:

- *liblinear-2.1* - složka obsahující knihovnu liblinear

Dále obsahuje soubory:

- `config_tr.py`, `config_te.py` - nastavení parametrů pro trénování a test
- `cp2tform.py` - funkce pro získání transformační matice
- `derivebinaryfeat.py` - funkce pro získání lokálních znaků
- `drawshapes.py` - funkce pro vykreslení význačných bodů
- `flipshape.py` - funkce pro zrcadlové otočení význačných bodů
- `forwardAffineTransform.py` - funkce pro dopřednou afinní transformaci
- `getbbox.py` - funkce pro získání bbox
- `getproposals.py` - funkce pro výpočet úhlu a poloměru pro upravení umístění význačných bodů
- `globalprediction.py` - funkce pro globální predikci
- `globalregression.py` - funkce pro globální regresi
- `loadsamples.py` - funkce pro načtení dat
- `resetshape.py` - funkce pro resetování změn význačných bodů
- `rotatepoints.py` - funkce pro rotaci bodů
- `rotateshape.py` - funkce pro rotaci tvarů
- `scaleshape.py` - funkce pro změnu velikosti tvarů
- `test_model.py` - funkce pro testování
- `train_model.py` - funkce pro trénování
- `train_randomfs.py` - funkce pro vygenerování random forest
- `translateshape.py` - funkce pro posun význačných bodů

B Příloha: Návod ke spuštění

Aby aplikace fungovala je nutné si nainstalovat následující programy a knihovny:

- Python verze 2.7
- openCv verze 3.1.0
- liblinear 2.1. (přiloženo na CD)
- Knihovny v Pythonu: Numpy, Scipy, Matplotlib

Dále je nutné mít k dispozici trénovací a testovací dataset s obrázky, ke kterému je nutné vytvořit `*.txt` soubor obsahující cestu a názvy trénovacích/testovacích obrázku (např. /home/kattynka/dp/datasets/afw/jmeno_obrazků.jpg).
Také je nutné nastavit cestu ke knihovně `liblinear`, návod je k dispozici na webu [8].
Nakonec je nutné změnit cesty v souborech: `train_model.py` a `test_model.py`.

Spustit trénování je možné souborem `train_model.py`. Při prvním spuštění programu je nutné, aby tento soubor byl spuštěn jako první, protože testování potřebuje soubor `dataProTest.npy`, který je výstupem funkce v `train_model.py`.
Pokud máme k dispozici soubor `dataProTest.npy`, můžeme spustit testování pomocí `test_model.py`.